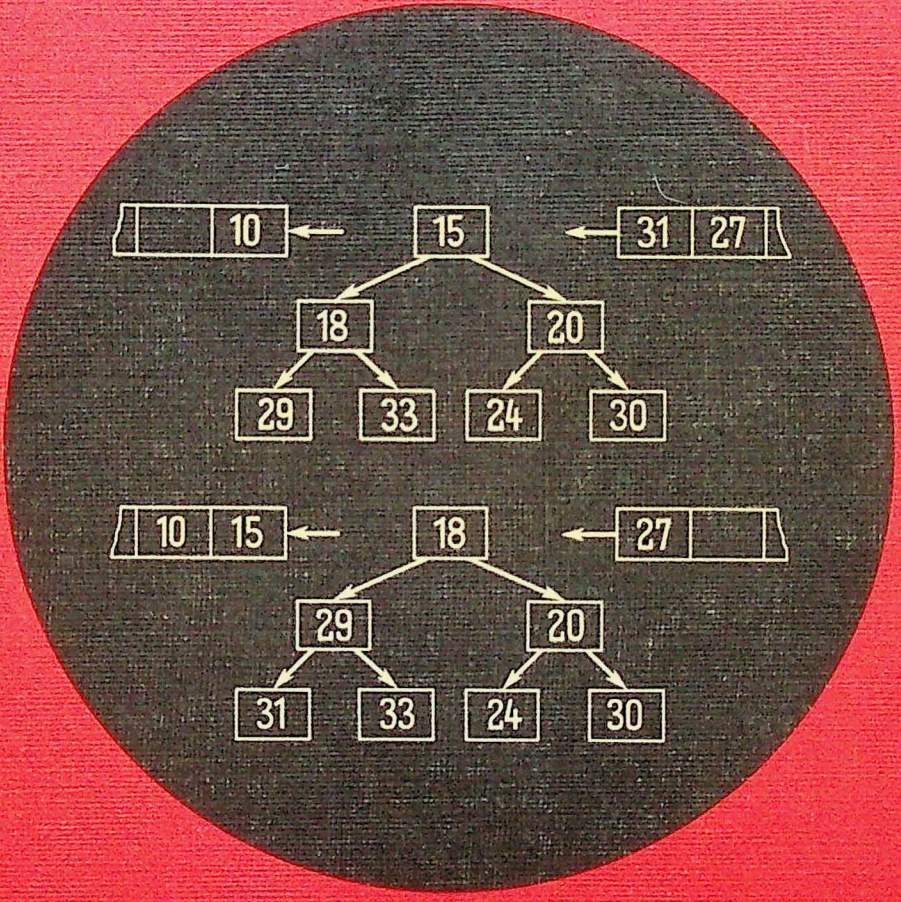
N. Wirth

Algorithmen und Datenstrukturen



Teubner Studienbücher
Informatik

Teubner Studienbücher

Informatik

Ehrig et al.: **Universal Theory of Automata**
240 Seiten. DM 22,80

Gilol: **Principles of Continuous System Simulation**
Analog, Digital and Hybrid Simulation in a Computer Science Perspective.
172 Seiten. DM 25,80 (LAMM)

Hotz: **Informatik: Rechenanlagen**
Struktur und Entwurf. 136 Seiten. DM 15,80 (LAMM)

Kandzia/Langmaack: **Informatik: Programmierung**
234 Seiten. DM 19,80 (LAMM)

Kupka/Wilsing: **Dialogsprachen**
168 Seiten. DM 19,80 (LAMM)

Maurer: **Datenstrukturen und Programmierverfahren**
222 Seiten. DM 25,80 (LAMM)

Schnorr: **Rekursive Funktionen und ihre Komplexität**
191 Seiten. DM 24,80 (LAMM)

Wirth: **Systematisches Programmieren**
Eine Einführung. 2. Aufl. 160 Seiten. DM 16,80 (LAMM)

Wirth: **Algorithmen und Datenstrukturen**
376 Seiten. DM 26,80 (LAMM)

Mathematik

Böhmer: **Spline-Funktionen**
Theorie und Anwendungen. 340 Seiten. DM 24,80

Clegg: **Variationsrechnung**
138 Seiten. DM 14,80

Collatz: **Differentialgleichungen**
Eine Einführung unter besonderer Berücksichtigung der Anwendungen.
5. Aufl. 226 Seiten. DM 21,80 (LAMM)

Collatz/Krabs: **Approximationstheorie**
Tschebyscheffsche Approximation mit Anwendungen. 208 Seiten. DM 26,80

Constantinescu: **Distributionen und ihre Anwendung in der Physik**
144 Seiten. DM 17,80

Fischer/Sacher: **Einführung in die Algebra**
238 Seiten. DM 15,80

Grigorieff: **Numerik gewöhnlicher Differentialgleichungen**
Band 1: Einschrittverfahren. 202 Seiten. 14,80 DM
Band 2: Mehrschrittverfahren

Hainzl: **Mathematik für Naturwissenschaftler**
311 Seiten. DM 29,– (LAMM)

Hilbert: **Grundlagen der Geometrie**
11. Aufl. VII, 271 Seiten. DM 19,80

Fortsetzung auf der 3. Umschlagseite

Teubner Studienbücher Informatik

N. Wirth
Algorithmen und Datenstrukturen

Leitfäden der angewandten Mathematik und Mechanik LAMM

Unter Mitwirkung von
Prof. Dr. E. Becker, Darmstadt
Prof. Dr. G. Hotz, Saarbrücken
Prof. Dr. K. Magnus, München
Prof. Dr. E. Meister, Darmstadt
Prof. Dr. Dr. h. c. F. K. G. Odqvist, Stockholm
Prof. Dr. Dr. h. c. Dr. h. c. E. Stiefel, Zürich

herausgegeben von
Prof. Dr. Dr. h. c. H. Görtler, Freiburg

Band 31

Die Lehrbücher dieser Reihe sind einerseits allen mathematischen Theorien und Methoden von grundsätzlicher Bedeutung für die Anwendung der Mathematik gewidmet; andererseits werden auch die Anwendungsgebiete selbst behandelt. Die Bände der Reihe sollen dem Ingenieur und Naturwissenschaftler die Kenntnis der mathematischen Methoden, dem Mathematiker die Kenntnisse der Anwendungsgebiete seiner Wissenschaft zugänglich machen. Die Werke sind für die angehenden Industrie- und Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler bestimmt, darüber hinaus aber sollen sie den im praktischen Beruf Tätigen zur Fortbildung im Zuge der fortschreitenden Wissenschaft dienen.

Algorithmen und Datenstrukturen

Von Dr. phil. N. Wirth
Professor an der Eidg. Technischen Hochschule
und Professor an der Universität Zürich

1975. Mit 93 Figuren, 30 Tabellen,
69 Übungen und zahlreichen Programmen

B. G. Teubner Stuttgart

Prof. Dr. phil. Niklaus Wirth

Geboren 1934 in Winterthur, Schweiz. Von 1954 bis 1958 Studium an der Eidg. Technischen Hochschule Zürich mit Abschluß als Dipl. El.-Ing. Von 1959 bis 1960 Studium an der Université Laval, Quebec, Canada, und Erlangung des Grades M. Sc. Von 1960 bis 1963 Studium und anschließend Promotion an der University of California, Berkeley. Von 1963 bis 1967 Assistant Professor of Computer Science an der Stanford University. Von 1967 bis 1968 Assistenzprofessor an der Universität Zürich. Seit 1968 Professor für Informatik an der Eidg. Technischen Hochschule und an der Universität Zürich.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Wirth , Niklaus
Algorithmen und Datenstrukturen : mit 93 Fig., 30 Tab., 69 Übungen u. zahlr. Programmen.
(Leitfäden der angewandten Mathematik und Mechanik; Bd. 31) (Teubner Studienbücher : Informatik)
ISBN 3-519-02330-X

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, besonders die der Übersetzung, des Nachdrucks, der Bildentnahme, der Funksendung, der Wiedergabe auf photomechanischem oder ähnlichem Wege, der Speicherung und Auswertung in Datenverarbeitungsanlagen, bleiben, auch bei Verwertung von Teilen des Werkes, dem Verlag vorbehalten.

Bei gewerblichen Zwecken dienender Vervielfältigung ist an den Verlag gemäß § 54 UrhG eine Vergütung zu zahlen, deren Höhe mit dem Verlag zu vereinbaren ist.

© B. G. Teubner, Stuttgart 1975

Printed in Germany
Druck: J. Beltz, Hemsbach/Bergstraße
Binderei: G. Gebhardt, Ansbach
Umschlaggestaltung: W. Koch, Sindelfingen

Unser Generalleutnant L.Euler gibt durch unsern Mund folgende Erklärung ab. Er bekennt offen:

III. dass er, obgleich Phönix der Algebristen, errötet und stets erröten wird über den Verstoss gegen die gesunde Vernunft und die gewöhnlichsten Begriffe, den er begangen hat, indem er aus seinen Formeln den Schluss zog, dass ein Körper, der von einem Zentrum aus angezogen wird, in der Mitte wieder umkehrt.

IV. dass er, um die deutschen Philosophen zu besänftigen, sein möglichstes tun wird, um künftig seine Vernunft nicht mehr an eine falsche Formel auszuliefern. Er bittet auf den Knien die Logiker um Verzeihung dafür, dass er einmal angesichts eines paradoxen Resultats den Satz geschrieben hat: 'wenn dies auch der Wahrheit zu widerstreiten scheint, so müssen wir doch der Rechnung mehr trauen als unserem Verstand'.

V. dass er künftig nicht mehr sechzig Seiten lang rechnen wird für ein Resultat, das man mit wenig Ueberlegung auf zehn Zeilen ableiten kann; und wenn er wieder seine Aermel aufkrempelt, um drei Tage und drei Nächte durchzurechnen, dass er dann vorher eine Viertelstunde zum Nachdenken verwenden will, welche Prinzipien am besten zur Anwendung kommen.

Auszug aus Voltaires
"Diatribe du docteur Akakia",
(November 1752)

Vorwort

In den vergangenen Jahren wurde die Programmierung von Rechenanlagen als die Disziplin erkannt, deren Beherrschung grundlegend und entscheidend für den Erfolg vieler Entwicklungsprojekte ist und die wissenschaftlicher Behandlung und Darlegung zugänglich ist. Vom Handwerk stieg sie zur akademischen Disziplin auf. Die ersten hervorragenden Beiträge zu dieser Entwicklung wurden von E.W. Dijkstra und C.A.R. Hoare geliefert. Dijkstra's "Notes on Structured Programming" [6] führten zu einer neuen Betrachtung der Programmierung als wissenschaftliches Thema und als intellektuelle Herausforderung. Sie bahnten den Weg zu einer "Revolution" in der Programmierung [35]. Hoare's "An Axiomatic Basis for Computer Programming" [10] zeigte deutlich, dass Programme einer exakten Analyse nach mathematischen Grundsätzen zugänglich sind. Beide Artikel argumentieren überzeugend, dass viele Programmierfehler vermieden werden können, wenn man den Programmierern die Methoden und Techniken, die sie bisher intuitiv und oft unbewusst verwendeten, zur Kenntnis bringt. Diese Artikel konzentrierten sich auf die Aspekte des Aufbauens und Analysierens von Programmen oder, genauer ausgedrückt, auf die Struktur der durch die Programmtexte dargestellten Algorithmen. Es ist jedoch völlig klar, dass ein systematisches und wissenschaftliches Angehen der Konstruktion von Programmen seine grösste Wirkung im Fall grosser komplexer Programme hat, die komplizierte Datenmengen bearbeiten. Folglich muss eine Methodik des Programmierens auch alle Aspekte der Datenstrukturierung behandeln. Programme sind letztlich konkrete Formulierungen abstrakter Algorithmen, die sich auf bestimmte Darstellungen und Datenstrukturen stützen. Einen wesentlichen Beitrag, Ordnung in die verwirrende Vielfalt der Terminologie und Konzepte von Datenstrukturen zu bringen, leistete Hoare durch seine "Notes on Data Structuring" [13]. Es wurde klar, dass Entscheidungen über die Strukturierung der Daten nicht ohne Kenntnis der auf die Daten anzuwendenden Algorithmen getroffen werden können und dass umgekehrt die Struktur und Wahl der Algorithmen oft stark von der Struktur der zugrunde liegenden Daten abhängt. Kurz gesagt: Programmerstellung und Datenstrukturierung sind untrennbar ineinandergreifende Themen.

Dennoch beginnt dieses Buch mit einem Kapitel über Datenstrukturen.

Dafür gibt es zwei Gründe. Erstens hat man das intuitive Gefühl, dass Daten den Algorithmen vorangehen; man muss Objekte haben, bevor man Operationen auf sie anwenden kann. Zweitens - und das ist der unmittelbare Grund - geht dieses Buch davon aus, dass der Leser mit den grundlegenden Begriffen des Programmierens von Rechenanlagen vertraut ist. Einführende Programmierkurse konzentrieren sich jedoch traditionell und vernünftigerweise auf Algorithmen, die auf einfachen Datenelementen operieren. Somit scheint ein einführendes Kapitel über Datenstrukturen angebracht.

Im ganzen Buch und besonders im ersten Kapitel folgen wir der von Hoare [13] dargelegten Theorie und Terminologie, die in der Programmiersprache PASCAL realisiert wurde [31]. Das Wesentliche dieser Theorie ist, dass Daten zuerst Abstraktionen realer Phänomene darstellen und vorzugsweise als abstrakte, in üblichen Programmiersprachen nicht notwendigerweise vorhandenen Strukturen ausgedrückt werden. Im Prozess der Programmentwicklung wird die Darstellung der Daten schrittweise verfeinert - parallel zur Verfeinerung der Algorithmen - um sich mehr und mehr den durch ein verfügbares Programmiersystem gegebenen Möglichkeiten anzupassen [32]. Wir setzen daher eine Anzahl grundlegender Prinzipien zum Aufbau von Datenstrukturen voraus, die sogenannten fundamentalen Strukturen. Es ist sehr wichtig, dass diese Konstruktionen auf wirklichen Rechenanlagen sehr leicht zu implementieren sind. Denn nur in diesem Fall können sie als wirkliche Elemente einer Datendarstellung betrachtet werden, als die aus dem letzten Verfeinerungsschritt hervorgehenden "Moleküle" der Datenbeschreibung. Es sind dies der Verbund (record), das Feld (array) und die Menge (set). Es überrascht nicht, dass diese grundlegenden Aufbauprinzipien fundamentalen mathematischen Begriffen entsprechen.

Ein Grundprinzip dieser Theorie der Datenstrukturen ist die Unterscheidung zwischen fundamentalen und "höheren" Strukturen. Erstere sind die Moleküle - selbst aus Atomen aufgebaut - und dienen als Komponenten für die letzteren. Variablen einer fundamentalen Struktur ändern nur ihren Wert, aber niemals ihre Struktur und nie die Wertemenge, die sie annehmen können. Folglich bleibt die Grösse des Speichers, den sie belegen, konstant. "Höhere" Strukturen aber sind charakterisiert durch die Veränderung ihres Wertes und ihrer Struktur während der Ausführung eines Programms. Für ihre Implemen-

tation werden daher aufwendigere Techniken benötigt.

Die sequentielle Datei - einfach File oder Sequenz genannt - erscheint in dieser Klassifikation als ein Zwitter. Sie ändert zwar ihre Länge; diese Aenderung der Struktur ist aber trivialer Natur. Da das File eine wirklich grundlegende Rolle in praktisch allen Rechenanlagen spielt, wird es mit den fundamentalen Strukturen in Kapitel 1 behandelt.

Das zweite Kapitel behandelt Sortier-Algorithmen. Es zeigt eine Vielfalt verschiedener Methoden, die alle dem gleichen Zweck dienen. Mathematische Analysen einiger dieser Algorithmen zeigen die Vor- und Nachteile der Methoden und bringen dem Programmierer die Wichtigkeit der Analyse bei der Wahl einer guten Lösung für ein gegebenes Problem zum Bewusstsein. Die Aufteilung in Methoden zum Sortieren von Arrays und Methoden zum Sortieren von Files (oft internes und externes Sortieren genannt) zeigt den entscheidenden Einfluss der Darstellung der Daten auf die Wahl anwendbarer Algorithmen und auf ihre Komplexität. Dem Sortieren würde nicht soviel Platz gewidmet, wenn es nicht ein ideales Instrument für die Darlegung so vieler Prinzipien der Programmierung und von Situationen wäre, die auch bei vielen anderen Anwendungen auftreten. Es scheint oft, dass man einen ganzen Programmierkurs anhand von Sortierbeispielen aufbauen könnte.

Ein anderes Thema, das gewöhnlich in Einführungskursen keinen Platz findet, spielt in der Konzeption vieler algorithmischer Lösungen eine wichtige Rolle, nämlich die Rekursion. Das dritte Kapitel ist deshalb rekursiven Algorithmen gewidmet. Es wird gezeigt, dass Rekursion eine Verallgemeinerung der Wiederholung (Iteration) ist. Als solche ist sie ein wichtiges und umfassendes Konzept der Programmierung. In vielen Programmierkursen wird Rekursion leider an Beispielen gezeigt, die mittels Iteration besser gelöst werden. Statt dessen konzentriert sich Kapitel 3 auf mehrere Beispiele von Problemen, in denen Rekursion eine äusserst natürliche Formulierung einer Lösung erlaubt, während die Verwendung von Iteration zu undurchsichtigen und schwerfälligen Programmen führen würde. Die Klasse der Backtracking-Algorithmen erweist sich als ideale Anwendung der Rekursion, aber die offensichtlichsten Kandidaten für die Verwendung der Rekursion sind Algorithmen, die auf rekursiv

definierten Datenstrukturen operieren. Diese Fälle werden im letzten Kapitel behandelt, wozu das dritte Kapitel eine gute Grundlage bildet.

Kapitel 4 behandelt dynamische Datenstrukturen, das sind Daten, die während der Ausführung des Programms ihre Struktur ändern. Es wird gezeigt, dass rekursive Datenstrukturen eine wichtige Unterklasse der allgemein verwendeten dynamischen Strukturen sind. Obwohl eine unmittelbar rekursive Definition in diesen Fällen sowohl natürlich als auch möglich ist, wird sie gewöhnlich in der Praxis doch nicht verwendet. Statt dessen wird der zu ihrer Implementation verwendete Mechanismus dem Programmierer bewusst gemacht, indem man ihn zwingt, explizite Referenz- oder Zeiger-Variablen zu verwenden. Dieses Buch folgt dieser Praxis und spiegelt den gegenwärtigen Stand der Technik wieder: Kapitel 4 ist der Programmierung mit Zeigern, Listen, Bäumen und Beispielen mit komplexeren Datengebilden gewidmet. Es zeigt die (etwas unpassend) sogenannte "Listenverarbeitung". Ein beträchtlicher Teil behandelt die Organisation von Bäumen und besonders das Durchsuchen von Bäumen. Das Kapitel schliesst mit einer Darstellung der gestreuten Speicherung, auch "hash"-Code genannt, die oft Suchbäumen vorgezogen wird. Damit bietet sich eine Möglichkeit zum Vergleich zweier fundamental verschiedener Techniken für eine häufig vorkommende Anwendung.

Programmieren ist eine konstruktive Tätigkeit. Wie kann eine aufbauende, schöpferische Fähigkeit gelehrt werden? Eine Möglichkeit besteht im Herauskristallisieren elementarer Konstruktionsgesetze aus vielen Anwendungen und ihrer systematischen Darstellung. Aber Programmieren ist ein weites und vielfältiges Gebiet, das oft komplexe geistige Tätigkeit erfordert. Die Vorstellung, es je zu einem reinen "Unterrichten von Rezepten" zusammenfassen zu können, ist verfehlt. Aus unserem Arsenal von Unterrichtsmethoden bleibt nur die sorgfältige Auswahl und Darstellung von Musterbeispielen. Natürlich sollten wir nicht glauben, dass jedermann gleich viel aus dem Studium von Beispielen lernt. Wesentlich bei diesem Vorgehen ist, dass vieles dem Studenten, seinem Fleiss und seiner Intuition überlassen bleiben muss. Dies gilt ganz besonders für relativ schwierige und lange Programmbeispiele. Ihr Einschluss in dieses Buch geschah nicht zufällig. Längere Programme sind in der Praxis der "Normalfall" und eignen sich eher dafür, die schwer zu fassende

aber wesentliche Zutat, genannt Stil und methodische Struktur, aufzuzeigen. Sie sollen auch als Uebung für das Lesen von Programmen dienen, das neben dem Schreiben zu oft vernachlässigt wird. Dies ist ein wichtiger Grund für den Einschluss grösserer vollständiger Programme als Beispiele. Der Leser wird durch eine allmähliche Entwicklung des Programms geführt; es werden ihm mehrere "Schnappschüsse" der Entstehung des Programms gezeigt, wobei sich diese Entwicklung als schrittweise Verfeinerung der Einzelheiten erweist. Ich halte es für wichtig, dass Programme in einer letzten Form unter hinreichender Berücksichtigung der Einzelheiten gezeigt werden, denn beim Programmieren steckt der Teufel im Detail. Obwohl eine Beschränkung der Ausführung auf das Prinzip eines Algorithmus und seine mathematische Analyse unter Ausschluss technisch bedingter Details für einen akademischen Geist anregend und herausfordernd wirken kann, vermag dieses Vorgehen den Praktiker nicht zu befriedigen. Ich habe mich daher strikt an die Regel gehalten, die Programme zum Schluss in einer Sprache anzugeben, in der sie auf einer Rechenanlage direkt ausgeführt werden können.

Damit stellt sich natürlich das Problem, eine Form zu finden, die auf einer Rechenanlage ausführbar ist und gleichzeitig doch soweit maschinenunabhängig bleibt, wie es von einem Lehrbuch gefordert werden muss. In dieser Beziehung erwiesen sich weder weitverbreitete Sprachen noch abstrakte Notationen als geeignet. Die Sprache PASCAL hingegen stellt einen geeigneten Kompromiss dar: sie wurde mit genau diesem Ziel entwickelt und daher auch in diesem Buch verwendet. Die Programme können von Programmierern leicht verstanden werden, die mit irgendeiner anderen höheren Programmiersprache, wie ALGOL 60 oder PL/I vertraut sind. Dies heisst aber nicht, dass eine vorangehende Einführung nicht von Nutzen wäre. Das Buch "Systematisches Programmieren" [33] vermittelt eine ideale Grundlage, da es sich auch auf die PASCAL-Notation stützt. Das vorliegende Werk ist nicht als Handbuch für die Sprache PASCAL gedacht; zu diesem Zweck sei der Leser auf die einschlägige Literatur verwiesen [15].

Dieses Buch ist eine Zusammenfassung - und gleichzeitig eine Ausarbeitung - verschiedener Programmierkurse, die an der Eidgenössischen Technischen Hochschule (ETH) Zürich gehalten wurden. Ich verdanke viele in diesem Buch dargelegten Ideen und Ansichten Diskussionen mit meinen Mitarbeitern an der ETH. Zu erwähnen sind

auch der stimulierende Einfluss der Treffen der IFIP Arbeitsgruppen 2.1 und 2.3 und besonders die bei diesen Gelegenheiten geführten Gespräche mit E.W. Dijkstra und C.A.R. Hoare. Zum Schluss möchte ich nicht vergessen, der ETH für die grosszügige Bereitstellung der Rechenmöglichkeiten zu danken, ohne die die Ausarbeitung dieses Lehrbuchs unmöglich gewesen wäre. Ich möchte all denen, die direkt oder indirekt zu diesem Buch beigetragen haben, meinen herzlichen Dank aussprechen.

Der vorliegende Text wurde mit Hilfe des Computers erstellt und editiert, wobei der ISO Zeichensatz verwendet wurde. Da somit weder Höher- noch Tieferstellen zur Verfügung steht, wird zur Angabe der Exponentiation das Symbol ** verwendet, und Indizes werden in eckige Klammern gesetzt.

Sowohl die sehr gelungene Uebersetzung aus dem Englischen Originaltext als auch die computertechnische Herstellung des druckreifen Textes sind das Werk von Dr. Helmut Sandmayr. Ihm gebührt für seine ausgezeichnete Arbeit mein ganz besonderer Dank.

Zürich, im Sommer 1975 N. Wirth

Inhalt

1. Fundamentale Datenstrukturen 15

1.1. Einleitung 15
1.2. Der Begriff des Datentyps 19
1.3. Einfache Datentypen 23
1.4. Einfache Standard-Typen 25
1.5. Unterbereich-Typen 29
1.6. Die Strukturart "Array" 30
1.7. Die Strukturart "Record" 36
1.8. Die Strukturart des varianten Record 42
1.9. Die Strukturart "Menge" (Set) 45
1.10. Darstellung von Array- Record- und Mengen-Strukturen 52
1.10.1. Darstellung von Arrays 53
1.10.2. Darstellung von Records 56
1.10.3. Darstellung von Mengen 58
1.11. Die Struktur des sequentiellen Files 59
1.11.1. Elementare File-Operatoren 64
1.11.2. Files mit Unterstrukturen 67
1.11.3. Texte 70
1.11.4. Ein Programm zum Aendern eines Files 79

2. Sortieren 88

2.1. Einleitung 88
2.2. Sortieren von Arrays 91
2.2.1. Sortieren durch direktes Einfügen 92
2.2.2. Sortieren durch direktes Auswählen 96
2.2.3. Sortieren durch direktes Austauschen 99
2.2.4. Sortieren durch Einfügen mit abnehmender Schrittweite 103
2.2.5. Sortieren mit Bäumen 107
2.2.6. Sortieren durch Zerlegen 113
2.2.7. Bestimmung des mittleren Elementes 121
2.2.8. Ein Vergleich der Sortiermethoden mit Arrays 124
2.3. Sortieren sequentieller Files 128
2.3.1. Direktes Mischen 128
2.3.2. Natürliches Mischen 135
2.3.3. Ausgeglichenes N-Weg-Mischen 143
2.3.4. Mehrphasen-Sortieren 151
2.3.5. Verteilung der ursprünglichen Läufe 166

3. Rekursive Algorithmen 176

3.1. Einleitung 176
3.2. Wo Rekursion zu vermeiden ist 179
3.3. Zwei Beispiele rekursiver Programme 183
3.4. Backtracking Algorithmen 191
3.4.1. Das Problem der acht Damen 198
3.4.2. Das Problem der stabilen Heirat 204
3.4.3. Das optimale Auswahlproblem 213

4. Dynamische Informationsstrukturen 222

4.1. Rekursive Datentypen 222
4.2. Zeiger oder Referenzen 226
4.3. Lineare Listen 233
4.3.1. Grundoperationen 233
4.3.2. Geordnete Listen und Neuordnung von Listen 238
4.3.3. Eine Anwendung: Topologisches Sortieren 248
4.4. Baumstrukturen 257
4.4.1. Grundlegende Operationen und Definitionen 257
4.4.2. Elementare Operationen auf binären Bäumen 268
4.4.3. Durchsuchen eines Baumes und Einfügen in einen Baum 272
4.4.4. Löschen in Bäumen 283
4.4.5. Analyse des Durchsuchens eines Baumes und Einfügens in einen Baum 284
4.4.6. Ausgeglichene Bäume 289
4.4.7. Einfügen in ausgeglichene Bäume 291
4.4.8. Löschen in ausgeglichenen Bäumen 298
4.4.9. Optimale Suchbäume 303
4.4.10. Ausgabe einer Baumstruktur 311
4.5. Vielweg-Bäume 323
4.5.1. B-Bäume 325
4.5.2. Binäre B-Bäume 340
4.6. Schlüssel-Transformationen (hashing) 349
4.6.1. Wahl einer Transformationsfunktion H 351
4.6.2. Behandlung der Kollision 352
4.6.3. Analyse der Schlüsseltransformation 358

Anhang A. Der ASCII-Zeichensatz 370

Anhang B. Verzeichnis der Programme 371

Literatur 372

Sachverzeichnis 375

1. Fundamentale Datenstrukturen

1.1. Einleitung

Der moderne Digital-Computer wurde entwickelt, um komplizierte und zeitraubende Berechnungen zu erleichtern und zu beschleunigen. Bei den meisten Anwendungen spielt seine Fähigkeit, grosse Mengen von Information zu speichern und wieder zugänglich zu machen, die wichtigste Rolle (und wird als Haupteigenschaft betrachtet); seine Fähigkeit zu rechnen, d.h. zu kalkulieren, Arithmetik auszuführen, ist in vielen Fällen nahezu belanglos.

In allen diesen Fällen stellt die grosse Menge an Information, die in irgendeiner Weise verarbeitet werden muss, eine Abstraktion eines Teils der realen Welt dar. Die der Rechenanlage zur Verfügung stehende Information besteht aus einer ausgewählten Menge von Daten über die reale Welt, nämlich der für das vorliegende Problem als wichtig erachteten Menge, von der man annimmt, dass damit die gewünschten Resultate erzielt werden können. Die Daten stellen eine Abstraktion der Wirklichkeit dar, weil die für dieses bestimmte Problem nebensächlichen und belanglosen Eigenschaften und Besonderheiten der realen Objekte unberücksichtigt bleiben. Eine Abstraktion ist somit auch eine Vereinfachung der Tatsachen.

Als Beispiel können wir die Personalkartei eines Arbeitgebers betrachten. Jeder Angestellte ist in dieser Kartei (abstrahiert) durch eine Menge von Daten vertreten, die für den Arbeitgeber, bzw. für seine Abrechnungen wichtig sind. Diese Daten enthalten einige Kennzeichen des Arbeitnehmers, wie z.B. seinen Namen und sein Gehalt. Sehr wahrscheinlich werden jedoch unwichtige Angaben, wie Haarfarbe, Gewicht und Grösse, nicht vermerkt sein.

Bei der Lösung eines Problems mit oder ohne Rechenanlage ist es notwendig, eine Abstraktion der Wirklichkeit zu wählen, d.h. eine Menge von Daten zur Darstellung der realen Situation zu definieren. Diese Wahl ist je nach dem zu lösenden Problem zu treffen. Danach folgt die Festlegung der Darstellung dieser Information, wobei das

Instrument zu berücksichtigen ist, mit dem das Problem gelöst werden soll. In den meisten Fällen sind die Auswahl der relevanten Information und die Festlegung einer dem Rechner angepassten Darstellung nicht völlig unabhängig voneinander.

Die Wahl der Darstellung der Daten ist oft ziemlich schwierig und wird nicht nur durch die vorhandenen Möglichkeiten bestimmt. Sie muss immer in bezug auf die mit den Daten durchzuführenden Operationen gesehen werden. Ein gutes Beispiel ist die Darstellung von Zahlen, die selbst Abstraktionen von Eigenschaften zu charakterisierender Objekte sind. Wenn die Addition die einzige (oder zumindest die überwiegend) auszuführende Operation ist, kann die Zahl n gut durch n Striche dargestellt werden. Die Additionsregel für diese Darstellung ist wirklich offensichtlich und einfach. Die römischen Zahlen basieren auf dem gleichen Prinzip, und die Additionsregeln sind für kleine Zahlen entsprechend einfach. Die Darstellung durch arabische Ziffern erfordert aber Regeln, die (für kleine Zahlen) bei weitem nicht offensichtlich sind und gelernt werden müssen. Die Situation kehrt sich jedoch um, wenn die Addition grosser Zahlen oder die Multiplikation und Division betrachtet werden. Die Zerlegung dieser Operationen in einfachere ist bei der Darstellung durch arabische Ziffern viel leichter wegen ihres systematischen Strukturprinzips, das auf dem Stellenwert der einzelnen Ziffern basiert. Es ist bekannt, dass Rechenanlagen eine auf binären Ziffern (Bits) basierende Darstellung benutzen. Diese Darstellung ist für Menschen wegen der üblicherweise grossen Anzahl vorkommender Ziffern ziemlich ungeeignet, sie eignet sich jedoch für elektronische Schaltungen, da sich die zwei Werte 0 und 1 leicht und zuverlässig durch An- oder Abwesenheit elektrischer Ströme, elektrischer Ladungen und magnetischer Felder darstellen lassen.

Aus diesem Beispiel entnehmen wir, dass die Frage der Darstellung oft mehrere Detailstufen umfasst. Wird das Problem gestellt, z.B. die Position eines Objekts darzustellen, so kann in der ersten Entscheidung ein Paar reeller Zahlen in kartesischen oder polaren Koordinaten gewählt werden. Die zweite Entscheidung kann zu einer Gleitkommadarstellung führen, wobei jede reelle Zahl x aus einem Paar <f,e> ganzer Zahlen besteht. Die Zahl f bezeichnet einen Bruch und e einen Exponenten zu einer bestimmten Basis (z.B. x = f*2**e).

Auf der dritten Stufe basiert die Entscheidung auf der Kenntnis, dass die Daten in einer Rechenanlage gespeichert werden. Dies kann zu einer binären positionsabhängigen Darstellung ganzer Zahlen führen. Die letzte Entscheidung könnte sein, binäre Ziffern durch die Richtung des magnetischen Flusses in einem Magnetkern darzustellen. Natürlich wird die erste Entscheidung in dieser Kette hauptsächlich durch die Problemstellung beeinflusst, und die darauffolgenden hängen dann von dem Gerät und dessen Technologie ab. Deshalb kann von einem Programmierer kaum verlangt werden, dass er über die zu verwendende Zahlendarstellung oder gar über die Eigenschaften der Speichervorrichtung entscheidet. Diese "Entscheidungen auf unterster Ebene" können den Entwicklern von Rechenanlagen überlassen werden. Diese sind über die moderne Technologie bestens informiert und können eine vernünftige und brauchbare Wahl für alle (oder die meisten) Anwendungen treffen, bei denen Zahlen eine Rolle spielen.

In diesem Zusammenhang wird die Bedeutung der Programmiersprachen offensichtlich. Eine Programmiersprache stellt einen abstrakten Computer dar, der Ausdrücke dieser Sprache verstehen kann. Die Elemente dieser Sprache verkörpern einen gewissen Grad an Abstraktion über die von der wirklichen Maschine benutzten Objekte. Daher werden dem Programmierer, der eine solche "höhere" Sprache benutzt, die Fragen der Zahlendarstellung abgenommen (d.h. Information über die verwendete Darstellung ist ihm gar nicht zugänglich), falls die Zahl ein Elementarobjekt im Bereich dieser Sprache ist.

Die Verwendung einer Sprache, die eine für die meisten bei der Datenverarbeitung auftretenden Probleme geeignete Menge grundlegender Abstraktionen bietet, ist hinsichtlich der Zuverlässigkeit der entstehenden Programme besonders wichtig. Es ist einfacher, ein Programm auszuarbeiten, das auf einer Beweisführung mit üblichen Begriffen wie Zahlen, Mengen, Folgen und Wiederholungen basiert, als auf rechnerorientierten Begriffen wie Bits, "Wörtern" und Sprüngen. Natürlich stellt eine wirkliche Rechenanlage alle diese Daten, ob Zahlen, Mengen oder Folgen, letztlich durch binäre Ziffern (Bits) dar. Aber dies ist für den Programmierer ziemlich nebensächlich, solange er sich nicht um die Einzelheiten der Darstellung seiner ausgewählten Abstraktionen kümmern muss, und solange er sicher sein kann, dass die entsprechende, vom Computer (oder Compiler) gewählte Darstellung für seine Zwecke richtig ist.

Je näher die Abstraktionen bei einer bestimmten Rechenanlage liegen, um so einfacher ist es für den Ingenieur oder den Entwickler eines Compilers, für die Sprache eine Wahl der Darstellung zu treffen, und um so grösser ist die Wahrscheinlichkeit, dass eine einzige Wahl für alle (oder fast alle) denkbaren Anwendungen geeignet ist. Diese Tatsache setzt dem Grad an Abstraktion von einem bestimmten wirklichen Computer feste Grenzen. Es wäre z.B. Unsinn, geometrische Objekte als grundlegende Datenelemente in eine allgemein verwendbare Sprache aufzunehmen, da deren genaue Darstellung wegen der ihnen eigenen Komplexität weitgehend von den auf diese Objekte anzuwendenden Operationen abhängt. Die Art und Häufigkeit dieser Operationen wird jedoch dem Entwickler einer allgemein verwenbaren Sprache und deren Compiler nicht bekannt sein, und jede von ihm getroffene Wahl kann für einige mögliche Anwendungen ungeeignet sein.

Diese Ueberlegungen bestimmen die Wahl der Notation für die Beschreibung von Algorithmen und ihrer Daten in diesem Buch. Genau gesagt, möchten wir bekannte mathematische Begriffe wie Zahlen, Mengen, Folgen, usw. benutzen und nicht vom Computer abhängige Einheiten wie Bit-Folgen. Ausserdem wollen wir eine Notation verwenden, für die tatsächlich brauchbare Compiler bereits existieren. Es ist ebenso unklug, eine streng maschinenorientierte und maschinenabhängige Sprache anzuwenden, wie es nutzlos ist, Computerprogramme in einer abstrakten Notation unter völliger Vernachlässigung von Darstellungsproblemen zu beschreiben.

Die Programmiersprache PASCAL wurde in einem Versuch entwickelt, einen Kompromiss zwischen diesen Extremen zu finden, und sie wird im ganzen vorliegenden Buch verwendet [15,31]. Sie wurde erfolgreich auf mehreren Rechenanlagen eingeführt, und es hat sich gezeigt, dass die Notation nahe genug bei den wirklichen Maschinen liegt, um die gewählten Elemente und ihre Darstellungen deutlich erklären zu können. Die Sprache ist auch nahe genug bei anderen Sprachen, besonders bei ALGOL 60, so dass die hier gelehrten Lektionen für deren Gebrauch direkt übernommen werden können.

1.2. Der Begriff des Datentyps

In der Mathematik ist es üblich, Variable nach bestimmten wichtigen Eigenschaften zu ordnen. Genaue Unterscheidungen werden gemacht zwischen reellen, komplexen und logischen Variablen oder zwischen Variablen, die einzelne Werte oder Wertemengen oder Mengen von Mengen darstellen, oder zwischen Funktionen, Funktionalen, Mengen von Funktionen, usw. Dieser Begriff der Klassifizierung ist in der Datenverarbeitung ebenso wichtig, wenn nicht sogar noch wichtiger. Wir werden uns nach dem Prinzip richten, wonach jede Konstante oder Variable, jeder Ausdruck oder jede Funktion von einem bestimmten Typ ist. Dieser Typ bezeichnet hauptsächlich die Wertemenge, der eine Konstante angehört, oder die durch eine Variable oder einen Ausdruck angenommen oder durch eine Funktion berechnet werden kann.

In mathematischen Texten ist der Typ einer Variablen gewöhnlich vom Schrifttyp ohne Beachtung des Zusammenhangs ableitbar, in Computerprogrammen ist dies nicht möglich, da eine Rechenanlage gewöhnlich nur einen Schrifttyp (lateinische Buchstaben) zur Verfügung stellt. Weitverbreitet ist deshalb die Regel, dass der verwendete Typ in einer Vereinbarung (declaration) der Konstanten, Variablen oder Funktion festgelegt wird und dass diese Vereinbarung im Text vor der Verwendung dieser Konstanten, Variablen oder Funktion steht. Diese Regel ist besonders verständlich unter Berücksichtigung der Tatsache, dass ein Compiler eine Wahl zur Darstellung des Objekts im Speicher des Computers zu treffen hat. Natürlich ist die einer Variablen zugeordnete Speichermenge je nach der Grösse des Wertebereichs zu wählen, den die Variable annehmen kann. Wenn diese Information dem Compiler bekannt ist, kann eine sogenannte dynamische Speicherzuordnung vermieden werden; das ist sehr oft der Schlüssel zu einer effizienten Realisierung eines Algorithmus.

Die hauptsächlichen Eigenschaften des Begriffs Typ, wie er in dem vorliegenden Text benutzt wird und wie er in der Programmiersprache PASCAL verkörpert ist, lassen sich folgendermassen zusammenfassen [13]:

1. Ein Datentyp bestimmt die Wertemenge, zu der eine Konstante gehört oder die durch eine Variable oder einen Ausdruck angenommen

oder die durch einen Operator oder eine Funktion berechnet werden kann.

2. Der Typ eines durch eine Konstante, Variable oder Ausdruck bezeichneten Wertes kann seiner Denotation oder seiner Vereinbarung entnommen werden, ohne den Rechenprozess durchführen zu müssen;

3. Jeder Operator oder jede Funktion erwartet Argumente eines bestimmten Typs und liefert ein Resultat eines bestimmten Typs. Wenn ein Operator Argumente verschiedener Typen zulässt (z.B. wird + zur Addition sowohl von ganzen als auch reellen Zahlen benutzt), dann kann der Typ des Ergebnisses nach bestimmten Sprachregeln ermittelt werden.

Folglich kann ein Compiler diese Typeninformation zur Prüfung der Vereinbarkeit und Zulässigkeit verschiedener Konstruktionen benutzen. Zum Beispiel kann die Zuweisung eines Booleschen (logischen) Wertes zu einer arithmetischen (reellen) Variablen ohne Ausführung des Programms entdeckt werden. Diese Art von Redundanz im Programmtext ist eine überaus grosse Hilfe bei der Programmentwicklung und muss als der wesentliche Vorteil guter höherer Sprachen gegenüber dem Maschinen-Code (oder dem symbolischen Assembler-Code) betrachtet werden. Natürlich werden im Endeffekt die Daten durch eine grosse Zahl binärer Ziffern dargestellt, unabhängig davon, ob das Programm ursprünglich in einer höheren, den Begriff des Typs benutzenden Sprache oder in einem typenlosen Assembler-Code betrachtet wurde. Für die Rechenanlage ist der Speicher eine homogene Masse von Bits ohne sichtbare Struktur. Es ist jedoch einzig und allein diese abstrakte Struktur, die es dem menschlichen Programmierer ermöglicht, einen Sinn in der monotonen Welt eines Computerspeichers zu finden.

Die in diesem Buch dargelegte Theorie und die Programmiersprache PASCAL beschreiben gewisse Methoden zur Definition von Datentypen. In den meisten Fällen werden neue Datentypen mit Hilfe vorher eingeführter Typen definiert. Werte eines solchen Typs sind gewöhnlich Zusammenfassungen von Komponentenwerten früher definierter Komponententypen und werden als strukturiert bezeichnet. Wenn es nur einen Komponententyp gibt, d.h. wenn alle Komponenten vom gleichen

Typ sind, dann heisst dieser Grundtyp, und die Struktur selbst wird als homogen bezeichnet.

Die Anzahl bestimmter, zu einem Typ T gehörender Werte heisst Kardinalität von T. Sie ist ein Mass für den Speicherplatz, der zur Darstellung einer Variablen x vom Typ T entsprechend der Notation

x: T

benötigt wird.

Da Komponententypen wieder strukturiert sein können, ist es möglich, ganze Strukturhierarchien zu erstellen. Es ist jedoch klar, dass die letzten Komponenten einer Struktur unteilbar sein müssen. Deshalb ist eine Notation zur Einführung solcher primitiver, unstrukturierter Typen vorzusehen. Eine direkte Methode ist die Aufzählung der den Typ bildenden Werte. In ein Programm mit einfachen geometrischen Figuren kann z.B. ein primitiver Typ, Form, eingeführt werden, dessen Werte durch die Namen Rechteck, Quadrat, Ellipse, Kreis bezeichnet werden können. Neben solchen vom Programmierer definierten Typen muss es einige Standard-Typen geben, die man als vordefiniert auffassen kann. Sie umfassen normalerweise Zahlen und logische Werte. Wenn eine Ordnung zwischen den einzelnen Werten besteht, wird der Typ geordnet oder skalar genannt. In PASCAL werden alle unstrukturierten Typen als geordnet angenommen; im Fall von expliziter Aufzählung entspricht die Ordnung der Reihenfolge der Aufzählung.

Mit diesem Instrument ist es möglich, primitive Typen zu definieren und Konglomerate zu bilden, d.h. strukturierte Typen bis zu einem willkürlichen Grad zu verschachteln. In der Praxis genügt eine allgemeine Methode allein nicht, um Komponententypen in einer Struktur zu vereinigen. Mit gebührender Rücksicht auf praktische Probleme bei Darstellung und Gebrauch muss eine allgemein verwendbare Programmiersprache verschiedene Methoden zur Strukturierung anbieten. Im mathematischen Sinn können sie alle gleichwertig sein; sie unterscheiden sich jedoch in den zur Verfügung stehenden Operatoren zur Bildung ihrer Werte und zur Auswahl von Komponenten dieser Werte. Die grundlegenden, hier dargestellten Strukturen sind der Array (Feld), der Record (Verbund), der Set (Menge) und das

File (Sequenz). Kompliziertere Strukturen werden gewöhnlich nicht als "statische" Typen definiert, sondern während der Ausführung des Programms "dynamisch" generiert; dabei können sie in Grösse und Form variieren. Solche Strukturen werden in Kapitel 4 behandelt und umfassen Listen, Ringe, Bäume und allgemeine endliche Graphen.

Variable und Datentypen werden in einem Programm eingeführt, weil sie bei einer Berechnung benutzt werden sollen. Zu diesem Zweck muss auch eine Menge von Operatoren zur Verfügung stehen. Wie bei Datentypen bieten Programmiersprachen eine gewisse Anzahl primitiver Standard-Operatoren und Methoden zur Strukturierung, durch die zusammengesetzte Operationen mittels primitiver Operatoren definiert werden können. Die Aufgabe des Zusammensetzens von Operationen wird oft als Kern der Programmierkunst bezeichnet. Es wird sich jedoch zeigen, dass die zweckmässige Anordnung der Daten ebenso fundamental und wichtig ist.

Die wichtigsten Grundoperationen sind Vergleich und Zuweisung, d.h. der Test auf Gleichheit (und Ordnung bei geordneten Typen) und der Befehl, Gleichheit zu erzwingen. Die grundlegende Differenz zwischen diesen beiden Operationen wird in diesem Text über die klare Unterscheidung ihrer Bezeichnung hervorgehoben (obwohl sie leider in so weitverbreiteten Programmiersprachen wie FORTRAN und PL/I durch die Verwendung des Gleichheitszeichens für den Zuweisungs-Operator verwischt wird).

Gleichheitstest: x =y
Zuweisung an x: x:=y

Diese fundamentalen Operatoren sind für ~~alle~~ viele Datentypen definiert. Es sollte jedoch beachtet werden, dass mit ihrer Ausführung ein beträchtlicher rechenmässiger Aufwand verbunden ist, wenn viele und stark strukturierte Daten vorliegen.

Neben diesem Test auf Gleichheit (oder Ordnung) und der Zuweisung gibt es eine Art fundamentaler und implizit definierter Operatoren, die sogenannten Transfer-Operatoren. Sie bilden Datentypen auf andere Datentypen ab. Besonders wichtig sind sie in Verbindung mit strukturierten Typen. Strukturierte Werte werden aus ihren Komponentenwerten durch sogenannte Konstruktoren generiert, und zu

den Komponentenwerten wird durch sogenannte Selektoren zugegriffen. Konstruktoren und Selektoren sind somit Transfer-Operatoren, die Komponententypen in strukturierte Typen abbilden und umgekehrt. Jede Strukturierungsmethode hat ihr eigenes Paar Konstruktor und Selektor, die sich deutlich durch ihre Notation unterscheiden.

Einfache Standard-Datentypen benötigen auch eine Menge einfacher Standard-Operatoren. Zusammen mit den Standard-Typen von Zahlen und logischen Werten führen wir deshalb auch die konventionellen Operationen von Arithmetik und Aussagenlogik ein.

1.3. Einfache Datentypen

In vielen Programmen werden ganze Zahlen zur Datendarstellung benutzt, auch wenn auf diese Daten keine numerischen Operatoren angewendet werden und die Variablen nur eine kleine Zahl von Alternativwerten annehmen können. In diesen Fällen führen wir einen neuen einfachen, unstrukturierten Datentyp T durch Aufzählung der Menge aller möglichen Werte c1, c2, ... , cn ein.

```
type T = (c1, c2, ... , cn)                    (1.1)
```

Die Kardinalität von T ist card(T) = n.

Beispiele:

```
type Form = (Rechteck, Quadrat, Ellipse, Kreis)
type Farbe = (rot, gelb, grün)
type Geschlecht = (männlich, weiblich)
type Boolean = (false, true)
type Wochentag = (Montag, Dienstag, Mittwoch, Donnerstag,
                  Freitag, Samstag, Sonntag)
type Währung = (Franken, Mark, Pfund, Doller, Schilling,
                Lire, Gulden, Krone, Rubel, Cruzeiro, Yen)
type Bestimmungsort = (Hölle, Fegefeuer, Himmel)
type Fahrzeug = (Zug, Bus, Auto, Schiff, Flugzeug)
```

```
type Rang = (Privatmann, Obergefreiter, Unteroffizier,
             Leutnant, Hauptmann, Major, Oberst, General)
type Objekt = (Konstante, Typ, Variable, Prozess, Funktion)
type Struktur = (file, array, record, set)
type Zustand = (manuell, ungeladen, paritÄt, schräglaufend)
```

Die Definition solcher Typen führt nicht nur einen neuen Namen ein, sondern gleichzeitig die ganze Menge der Namen, die die Werte des neuen Typs bilden. Diese Namen können dann als Konstanten im ganzen Programm benutzt werden und tragen wesentlich zu dessen Verständlichkeit bei. Wenn wir als Beispiel die Variablen s, d, r und b einführen

```
var s: Geschlecht
var d: Wochentag
var r: Rang
var b: Boolean
```

sind folgende Zuweisungen möglich:

```
s:= männlich
d:= Sonntag
r:= Major
b:= true
```

Offensichtlich sagen sie bedeutend mehr aus als ihre Gegenstücke

```
s := 1   d := 7   r := 6   b := 2 ,
```

die auf der Annahme beruhen, dass c, d, r und b vom Typ integer sind und dass die Konstanten in der Reihenfolge ihrer Aufzählung auf die natürlichen Zahlen abgebildet werden. Ausserdem kann ein Compiler prüfen, ob arithmetische Operatoren auf solch nicht-numerischen Typen ungewollt angewendet werden, wie z.B.

```
s := s+1 .
```

Wenn wir jedoch einen Typ als geordnet betrachten, so ist es vernünftig, solche Funktionen einzuführen, die den Nachfolger und Vorgänger ihres Arguments generieren. Diese Funktionen werden mit

succ(x) und pred(x) bezeichnet. Die Ordnung von Werten von T wird durch die Regel (1.2) festgelegt.

$$(ci<cj) = (i<j) \qquad (1.2)$$

1.4. Einfache Standard-Typen

Einfache Standard-Typen sind solche Typen, die in den meisten Computern fest eingebaut sind. Sie umfassen die ganzen Zahlen, die logischen Wahrheitswerte und eine Menge von Schriftzeichen. Auf grösseren Computern sind auch Brüche sowie eine passende Menge einfacher Operatoren verfügbar. Wir bezeichnen diese Typen mit den Namen

integer, Boolean, char, real .

Der Typ integer umfasst eine Teilmenge der ganzen Zahlen, deren Grösse bei den einzelnen Rechenanlagen verschieden ist. Es wird jedoch angenommen, dass alle Operationen mit Daten dieses Typs genau sind und den normalen arithmetischen Gesetzen entsprechen und dass der Rechenprozess unterbrochen wird, falls ein Ergebnis ausserhalb der darstellbaren Teilmenge liegt. Die Standard-Operatoren sind die vier arithmetischen Grundoperationen Addition (+), Subtraktion (-), Multiplikation (*) und Division (div). Von letzterer wird ein ganzzahliges Resultat erwartet, ein möglicher Rest wird nicht beachtet, so dass für positives m und n gilt

$$m-n < (m \text{ div } n)*n <= m \ . \qquad (1.3)$$

Der Modulo-Operator (mod) ist mit Hilfe der Division definiert durch die Gleichung

$$(m \text{ div } n)*n + (m \text{ mod } n) = m \qquad (1.4)$$

Somit ist m div n der ganzzahlige Quotient von m und n und m mod n der zugehörige Rest.

Der Typ real bezeichnet eine Teilmenge der reellen Zahlen. Während man von ganzzahliger Arithmetik annimmt, dass sie exakte Resultate liefert, sind bei Arithmetik mit Werten des Typs real Ungenauigkeiten innerhalb der Grenzen von Abrundungsfehlern erlaubt. Die Fehler werden durch Rechnen mit einer begrenzten Zahl von Ziffern verursacht. Dies ist der wesentliche Grund für die genaue Unterscheidung zwischen den Typen integer und real, die in den meisten Programmiersprachen gemacht wird.

Wir kennzeichnen die Division von reellen Zahlen, die einen reellwertigen Quotienten liefern, durch einen Schrägstrich (/) im Gegensatz zu div für die ganzzahlige Division.

Die beiden Werte des Standard-Typs Boolean werden durch die Namen true und false gekennzeichnet. Die Booleschen Operatoren sind die logische Konjunktion, Vereinigung und Negation, deren Werte in Tabelle 1.1 definiert sind. Die logische Konjunktion wird bezeichnet durch and, die logische Vereinigung durch or und die Negation durch not.

p	q	p or q	p and q	not p
T	T	T	T	F
T	F	T	F	F
F	T	T	F	T
F	F	F	F	T

Tabelle 1.1

Man beachte, dass Vergleiche Operatoren sind, die ein Resultat vom Typ Boolean liefern. Somit kann das Resultat eines Vergleichs einer Variablen zugewiesen werden, oder es kann als Operand eines logischen Operators in einem Booleschen Ausdruck verwendet werden. Sind zum Beispiel die Booleschen Variablen p und q und die ganzzahligen Variablen x = 5, y = 8 und z = 10 gegeben, so liefern die beiden Anweisungen

```
p := x = y
q := (x < y) and (y <= z)
```

die Resultate p = false und q = true.

Der Standard-Typ char bezeichnet eine Menge von Schriftzeichen. Leider gibt es keinen allgemein gültigen Standard-Zeichensatz, der auf allen Rechenanlagen verwendet wird. Deshalb ist der Gebrauch des Prädikats "Standard" in diesem Fall fast irreführend; es ist im Sinn von "Standard für die Rechenanlage, auf dem ein gewisses Programm auszuführen ist" zu verstehen.

Am weitesten verbreitet ist wohl der durch die International Standards Organization (ISO) definierte Zeichensatz und besonders seine amerikanische Version ASCII (American Standard Code for Information Interchange); der ASCII-Satz ist deshalb im Anhang aufgeführt. Er besteht aus 95 Schriftzeichen (graphic characters) und 33 Steuerzeichen (control characters), wobei letztere hauptsächlich bei der Datenübermittlung und für die Steuerung der Druckgeräte verwendet werden. Weitverbreitet ist eine Teilmenge von 64 Schriftzeichen (nur grosse Buchstaben), der sogenannte reduzierte ASCII-Satz (restricted ASCII character set).

Um Algorithmen angeben zu können, die zwar druckbare Zeichen enthalten (d.h. Werte vom Typ char), aber trotzdem von einer speziellen Rechenanlage unabhängig sein sollen, fordern wir von einem Zeichensatz gewisse verbindliche Eigenschaften, nämlich:

1. Der Typ char enthält die 26 lateinischen Buchstaben, die 10 arabischen Ziffern und eine Anzahl anderer graphischer Zeichen, wie z.B. Satzzeichen.

2. Die Teilmengen der Buchstaben und Ziffern sind geordnet und zusammenhängend, d.h.

   ('A' <= x) and (x <= 'Z') = x ist ein Buchstabe (1.5)
   ('0' <= x) and (x <= '9') = x ist eine Ziffer

3. Der Typ char enthält ein Leerzeichen, das als Trennzeichen verwendet werden kann (Leerzeichen werden in Fig. 1.1 durch _ dargestellt).

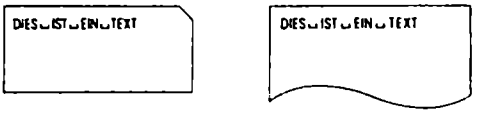


Fig. 1.1: Darstellung eines Textes mit Lochkarten und Liste

Das Vorhandensein von zwei Standard-Transferfunktionen zwischen den Typen char und integer ist besonders wichtig bei dem Versuch, Programme in einer maschinenunabhängigen Form zu schreiben. Wir werden sie ord(c) nennen für die Ordinalzahl des Zeichens c in der Zeichenmenge char und chr(i) für das i-te Zeichen in der Zeichenmenge char. Somit ist chr die Umkehrfunktion von ord und umgekehrt, d.h.

```
ord(chr(i)) = i (wenn chr(i) definiert ist)          (1.6)
chr(ord(c)) = c .
```

Besonders zu beachten sind die Funktionen

```
f(c) = ord(c) - ord('0') = Position von c innerhalb der Ziffern
g(i) = chr(i + ord('0')) = i-te Ziffer                (1.7)
```

Zum Beispiel gilt f('3') = 3 und g(5) = '5'. f ist die Umkehrfunktion von g und umgekehrt, d.h.

```
f(g(i)) = i          (0 <= i <= 9)                    (1.8)
g(f(c)) = c          ('0' <= c <= '9')
```

Diese Transferfunktionen werden bei der Umwandlung von internen Zahlendarstellungen in Ziffernfolgen und umgekehrt verwendet. Sie stellen tatsächlich diese Umwandlungen auf der elementarsten Stufe dar, nämlich der einer einzelnen Ziffer.

1.5. Unterbereich-Typen

In vielen Fällen nimmt eine Variable Werte eines bestimmten Typs nur innerhalb eines gewissen Intervalls an. Dies lässt sich durch Definition der Variablen von einem Unterbereich-Typ mit den Intervallgrenzen min und max ausdrücken:

```
type T = min..max                                    (1.9)
```

Beispiele:

```
type Jahr = 1900 .. 1999
type Buchstabe = 'A' .. 'Z'
type Ziffer = '0' .. '9'
type Offizier = Leutnant .. General
```

Zu den durch

```
var y: Jahr
var L: char
```

gegebenen Variablen sind die Zuweisungen y := 1973 und L := 'W' zulässig, jedoch y := 1291 und L := '9' sind es nicht. Die Gültigkeit einer solchen Zuweisung kann nicht vom Compiler geprüft werden, es sei denn, der zuzuweisende Wert wird durch eine Konstante oder eine Variable des gleichen Typs bezeichnet. Jedoch kann die Zulässigkeit von Zuweisungen wie

```
y := i   und   L := c ,
```

wobei i vom Typ integer und c vom Typ char ist, bei Ausführung des Programms geprüft werden. In der Praxis haben sich Systeme, die diese Prüfungen ausführen, bei der Programmentwicklung als besonders wertvoll erwiesen. Ihre Verwendung von redundanter Information, um mögliche Fehler zu finden, ist ein weiterer Grund für die Benutzung einer höheren Sprache.

1.6. Die Strukturart "Array"

Der "Array" ist wohl die am weitesten verbreitete Datenstruktur, da sie für viele Sprachen, auch FORTRAN und ALGOL 60, die einzige explizit verfügbare Struktur ist. Ein "Array" ist eine homogene Struktur; er besteht nur aus Komponenten des gleichen Typs, des sogenannten Grundtyps. Der Array ist auch eine sogenannte random-access Struktur; alle Komponenten können willkürlich zugegriffen werden und sind gleich schnell zugänglich. Um eine einzelne Komponente zu bezeichnen, wird der Name der gesamten Struktur um den sogenannten Index zur Wahl der Komponente erweitert. Dieser Index muss ein Wert des Typs sein, der als Indextyp des Array definiert ist. Die Definition eines Array-Typs T spezifiziert deshalb sowohl einen Komponententyp T0 als auch einen Indextyp I.

```
type T = array [I] of T0                                    (1.10)
```

Beispiele:

```
type Row = array [1..5] of real
type Card = array [1..80] of char
type alfa = array [1..10] of char
```

Ein besonderer Wert einer Variablen

```
var x: Row ,
```

deren Komponenten die Gleichung x[i] = 2**-i erfüllen, kann wie in Fig. 1.2 dargestellt werden.

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

Fig. 1.2: Array

Ein strukturierter Wert x vom Typ T mit Komponentenwerten c1, ... , cn kann durch einen Array-Konstruktor und eine Zuweisung bezeichnet werden:

$$x := T(c1, ..., cn) \qquad (1.11)$$

Der Umkehr-Operator des Konstruktors ist der Selektor. Er wählt eine einzelne Komponente aus dem Array aus. Bei einer Array-Variablen x bezeichnen wir einen Array-Selektor durch den Array-Namen mit dem Zusatz des entsprechenden Komponenten-Index i :

$$x[i] \qquad (1.12)$$

Beim Arbeiten mit Arrays, besonders mit grossen Arrays, ist es üblich, einzelne Komponenten selektiv zu verändern, anstatt ganz neue strukturierte Werte zu bilden. Dies wird dadurch ausgedrückt, dass eine Array-Variable als Darstellung eines Array von Komponenten-Variablen betrachtet wird und dass Zuweisungen an einzelne Komponenten erlaubt sind.

Beispiel:

$$x[i] := 0.125$$

Obwohl durch die selektive Anpassung nur ein einzelner Komponentenwert geändert wird, müssen wir vom begrifflichen Standpunkt aus auch den ganzen zusammengesetzten Wert als geändert ansehen.

Die Tatsache, dass Array-Indizes, d.h. "Namen" von Array-Komponenten, von einem definierten (skalaren) Datentyp sein müssen, hat eine äusserst wichtige Konsequenz:

Indizes können berechnet werden: ein Index-Ausdruck kann anstelle einer Index-Konstanten stehen. Dieser Ausdruck ist auszuwerten, und das Ergebnis bestimmt die Wahl der Komponente. Diese Allgemeinheit bietet nicht nur eine wichtige und spürbare Erleichterung beim Programmieren, sondern verursacht gleichzeitig einen der häufigst vorkommenden Programmierfehler: Der resultierende Wert kann ausserhalb des festgelegten Indexbereichs des Array liegen. Wir wollen annehmen, dass geeignete Systeme eine Warnung geben, wenn solch ein irrtümlicher Zugriff zu einer nicht existierenden Array-Komponente

versucht wird.

Normalerweise muss ein Indextyp ein skalarer Typ sein, d.h. ein unstrukturierter Typ, auf dem eine Ordnungsbeziehung definiert ist. Wenn der Komponententyp eines Array auch ein geordneter Typ ist, ist eine natürliche Ordnungsbeziehung für diesen Array-Typ gegeben. Die natürliche Ordnung von zwei Arrays wird durch die beiden entsprechenden, voneinander verschiedenen Komponenten mit den niedrigsten Indizes bestimmt. Formell wird dies wie folgt ausgedrückt:

```
x,y: array [I] of B                                    (1.13)
```

Es gilt x<y, wenn und nur wenn es ein k gibt, so dass
x[k] < y[k] und x[i] = y[i] für alle i<k.

Zum Beispiel gilt:

```
(2,3,5,7,9) < (2,3,5,7,11)
'SCHALE' < 'SCHULE' .
```

Die Kardinalität eines strukturierten Typs ist das Produkt der Kardinalität seiner Komponenten. Da alle Komponenten eines Array-Typs A vom gleichen Grundtyp B sind, erhalten wir

```
Kardinalität(A) = (Kardinalität(B))**n ,               (1.14)
```

wobei n= Kardinalität(I) und I der Indextyp des Array ist.

Das folgende kurze Programmstück zeigt die Verwendung des Array-Selektors. Aufgabe des Programms ist es, den kleinsten Index i einer Komponente mit dem Wert x zu finden. Die Suche wird durch ein sequentielles Durchlaufen des Array a ausgeführt und dargestellt als

```
var a: array [1..N] of T;                              (1.15)
i := 0;
repeat  i := i+1  until (a[i]=x) or (i=N);
if a[i] <> x then "es gibt kein solches Element in a"
```

Eine Variante dieses Programms verwendet die übliche Technik einer Marke (sentinel), die am Ende des Array steht. Diese Marke ermöglicht eine Vereinfachung der Schlussbedingung der Schleife.

```
var a: array [1..N+1] of T:                                    (1.16)
i := 0; a[N+1] := x;
repeat  i := i+1  until a[i] = x;
if i > N then "es gibt kein solches Element in a"
```

Die Zuweisung a[N+1] := x ist ein Beispiel für selektives Aendern, d.h. für die Veränderung einer speziellen Komponente einer strukturierten Variablen. Unabhängig von der Zahl der Wiederholungen der Anweisung i := i+1 ist die wesentliche Bedingung

```
a[j] <> x, für j= 1 ... i-1
```

in beiden Versionen (1.15) und (1.16) erfüllt. Diese Bedingung wird daher auch Schleifeninvariante genannt.

Die Suche kann natürlich bedeutend beschleunigt werden, wenn die Elemente bereits geordnet (sortiert) sind. In diesem Fall ist das Prinzip der wiederholten Halbierung des Intervalls, in dem das gewünschte Element gesucht werden muss, das gebräuchlichste. Es wird Bisektion oder binäre Suche genannt und ist in (1.17) dargestellt. Bei jeder Wiederholung wird das zu durchsuchende Intervall zwischen den Indizes i und j halbiert. Die Zahl der erforderlichen Vergleiche ist deshalb durch log2(N) beschränkt.

```
i := 1; j := N;                                                (1.17)
repeat k:= (i+j) div 2;
   if  x > a[k]  then  i := k+1  else  j := k-1
until (a[k]=x) or (i>j)
```

(Die relevante Invarianten-Bedingung zu Beginn der repeat-Anweisung ist

```
a[h] < x   für   h = 1 ... i-1
a[h] > x   für   h = j+1 ... N
```

Folglich bedeutet Beendigung des Programms mit i>j, dass es kein a[h] = x mit 1 <= h <= N gibt.)

Komponeneten von Array-Typen können selbst strukturiert sein. Eine Array-Variable, deren Komponenten ebenfalls Arrays sind, heisst Matrix. Zum Beispiel ist

```
M: array [1..10] of Row
```

ein aus zehn Komponenten (Zeilen) bestehender Array, jede einzelne Komponente besteht selbst aus fünf Komponenten vom Typ real. M heisst eine (10x5) Matrix mit reellen Komponenten. Selektoren können entsprechend aneinandergefügt werden, so dass

```
M[i][j]
```

die j-te Komponente der Zeile M[i] bezeichnet, während M[i] wiederum die i-te Komponente von M ist. Dies wird gewöhnlich abgekürzt zu

```
M[i,j]
```

und entsprechend kann die Vereinbarung

```
M: array [1..10] of array [1..5] of real
```

abgekürzt werden zu

```
M: array [1..10,1..5] of real .
```

Wenn eine bestimmte Operation für alle Komponenten eines Array oder mit aufeinanderfolgenden Komponenten eines Teils des Array durchgeführt werden soll, kann dies wie im folgenden Beispiel durch Verwendung der for-Anweisung entsprechend betont werden.

Angenommen ein Bruch f wird durch den Array d dargestellt, so dass

$$f = \sum_{i=1}^{k-1} d[i] * 10^{**-i}$$

d.h. in seiner Dezimalform mit k-1 Ziffern. Nun ist f durch 2 zu dividieren. Dies erfolgt durch Wiederholung der bekannten Divisionsoperation für alle k-1 Ziffern d[i], beginnend mit i=1. Man dividiert die Ziffer durch 2 unter Berücksichtigung eines möglichen Uebertrags von der vorhergehenden Position, und man bewahrt einen möglichen Rest r für den nächsten Schritt auf (siehe (1.18)).

```
r := 10 * r + d[i];
d[i] := r div 2;                                    (1.18)
r := r - 2*d[i]
```

Dieser Prozess wird in Programm 1.1 angewendet, um eine Tabelle der negativen Potenzen von 2 zu berechnen. Das wiederholte Halbieren zur Berechnung von 2 hoch -1, -2, ... , -n wird wiederum zweckmässig ausgedrückt durch eine for-Anweisung, woraus sich eine Verschachtelung von zwei for-Anweisungen ergibt.

```
program power(output);
{dezimale Darstellung der negativen Potenzen von 2}
const n = 10;
type digit = 0..9;
var i,k,r: integer;
    d: array [1..n] of digit;
begin  for k := 1 to n do
   begin write('.'); r := 0;
      for i := 1 to k-1 do
      begin r := 10*r + d[i]; d[i] := r div 2;
            r := r - 2*d[i]; write(chr(d[i]+ord('0')))
      end ;
      d[k] := 5; writeln('5')
   end
end.
```

Programm 1.1

Für n = 10 erhalten wir das folgende Resultat:

.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625

1.7. Die Strukturart "Record"

Die allgemeinste Methode zum Erstellen von strukturierten Typen ist das Zusammenfassen von Elementen willkürlicher, möglicherweise selbst strukturierter Typen zu einer Einheit. Beispiele aus der Mathematik sind die komplexen Zahlen, bestehend aus zwei reellen Zahlen, sowie Koordinaten von Punkten, die aus zwei oder mehr reellen Zahlen bestehen, je nach der Dimension des durch das Koordinatensystem aufgespannten Raumes. Ein Beispiel aus der Datenverarbeitung ist eine Personenbeschreibung durch wenige sachdienliche Eigenschaften, wie Vor- und Nachnamen, Geburtsdatum, Geschlecht und Zivilstand einer Person.

In der Mathematik heisst solch ein zusammengesetzter Typ das kartesische Produkt seiner Komponenten, weil die durch diesen zusammengesetzten Typ definierte Wertemenge alle möglichen Kombinationen von Werten umfasst, je einen aus jeder durch die Komponententypen definierten Menge. Somit ist die Zahl dieser Kombinationen, auch n-Tupel genannt, das Produkt der Anzahl von Elementen in jeder Teilmenge, d.h. die Kardinalität des zusammengesetzten Typs ist das Produkt der Kardinalitäten der Komponententypen.

In der Datenverarbeitung kommen zusammengesetzte Typen, wie Personen- oder Sachbeschreibungen, gewöhnlich in Files oder "Datenbanken" vor und registrieren die wichtigen Eigenschaften dieser Person oder dieses Objekts. Das Wort Record hat sich deshalb zur Beschrei-

bung eines Verbands von Daten dieser Art eingebürgert, und wir ziehen diese Bezeichnung dem Ausdruck "kartesisches Produkt" vor.

Ein Record-Typ T wird allgemein definiert wie folgt:

```
type T = record s1: T1;                                    (1.19)
                s2: T2;
                 ...
                sn: Tn
         end
```

Kardinalität(T) = Kardinalität(T1)* ... * Kardinalität(Tn)

Beispiele:

```
type Komplex = record re: real;
                      im: real
               end
(oder abgekürzt:  record re,im: real end  )

type Koordinate = record x: real;
                         y: real
                  end

type Datum = record tag: 1..31;
                  monat: 1..12;
                   jahr: 1..2000
             end

type Person = record name: alfa;
                    vorname: alfa;
                    geburtsdatum: Datum;
                    geschlecht: (männlich, weiblich);
                    zivilstand: (ledig, verheiratet,
                                 verwitwet, geschieden)
              end
```

Ein Wert vom Typ T kann durch einen Record-Konstruktor aus Werten xi vom Komponententyp Ti bezeichnet und einer Variablen dieses Typs zugewiesen werden:

```
x := T(x1,x2,...,xn)                                    (1.20)
```

Den Record-Variablen

```
z: Komplex
d: Datum
p: Person
```

können zum Beispiel folgende Werte zugewiesen werden
(vgl. Fig. 1.3)

```
z := Komplex(1.0,-1.0)
d := Datum(1,4,1973)
p := Person('Wirth','Chris',Datum(18,1,1966),
            männlich,ledig)
```

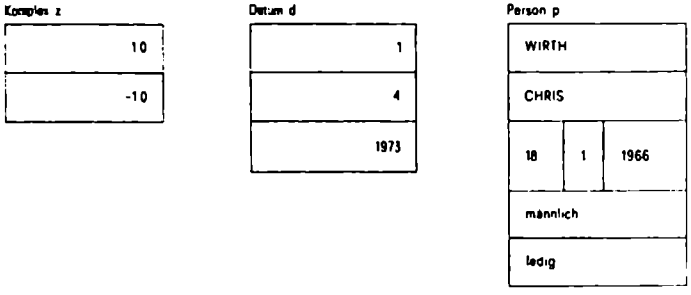


Fig. 1.3: Records

Die durch eine Record-Typdefinition eingeführten Bezeichner s1,..., sn sind die Namen, die den einzelnen Komponenten von Variablen jenes Typs gegeben werden, und sie werden in Record-Selektoren für Variable mit Record-Struktur verwendet. Gegeben sei eine Variable x:T, dann wird ihre i-te Komponente bezeichnet mit

```
x.si                                                    (1.21)
```

Die selektive Aenderung von x wird durch die gleiche Selektor-Schreibweise auf der linken Seite in einer Zuweisung erreicht:

```
x.si := xi
```

wobei xi einen Wert (Ausdruck) vom Typ Ti darstellt.

Zu Komponenten dieser Record-Variablen z, d und p kann durch folgende Selektoren zugegriffen werden:

```
z.im                    (vom Typ real)
d.monat                 (vom Typ 1..12)
p.name                  (vom Typ alfa)
p.geburtsdatum          (vom Typ Datum)
p.geburtsdatum.tag      (vom Typ 1..31)
```

Das Beispiel von Typ Person zeigt, dass eine Komponente eines Record-Typs selbst strukturiert sein kann. Deshalb können Selektoren aneinandergereiht werden. Natürlich können auch verschiedene Strukturtypen in einer verschachtelten Form verwendet werden. Zum Beispiel wird die i-te Komponente eines Array a, die eine Komponente einer Record-Variablen r ist, bezeichnet mit

```
r.a[i] ,
```

und die Komponente mit dem Selektor-Namen s der i-ten recordstrukturierten Komponente des Array a wird bezeichnet mit

```
a[i].s
```

Es ist für das kartesische Produkt charakteristisch, dass es alle Kombinationen von Elementen der Komponententypen umfasst. Es ist jedoch zu beachten, dass bei der praktischen Anwendung nicht alle "legal", d.h. sinnvoll sein können. Zum Beispiel enthält der oben definierte Typ Datum die Werte

```
(31,4,1973) und (29,2,1815) .
```

Beides sind Daten von Tagen, die es nie gab. Somit gibt die Definition dieses Typs nicht die Situation der wirklichen Welt wieder; sie genügt jedoch für praktische Zwecke, und es liegt im Verantwortungsbereich des Programmierers, dafür zu sorgen, dass während der Ausführung des Programms niemals sinnlose Werte auftreten.

Der kurze Programmauszug (1.22) zeigt die Verwendung von Record-

Variablen. Es sollen dabei die in der Array-Variablen a dargestellten Personen gezählt werden, die sowohl weiblich als auch ledig sind:

```
var a: array [1..N] of Person;                                    (1.22)
    zähler: integer;

zähler := 0;
for i := 1 to N do
   if (a[i].geschlecht = weiblich) and
                          (a[i].zivilstand=ledig) then
      zähler := zähler + 1
```

Die relevante Schleifeninvariante ist

```
zähler = Z(i)
```

wobei Z(i) die Zahl der ledigen weiblichen Personen der Untermenge a[1] ... a[i] ist.

Eine in der Notation verschiedene Variante der obigen Anweisung verwendet die sogenannte with-Anweisung:

```
for i := 1 to N do                                                (1.23)
    with a[i] do
       if (geschlecht = weiblich) and (zivilstand = ledig) then
          zähler:= zähler + 1
```

Dabei bedeutet with r do s, dass die Selektor-Namen des Typs der Record-Variablen r innerhalb der Anweisung s ohne Prefix verwendet werden und sich auf die Variable r beziehen. Durch die with-Anweisung wird somit der Programmtext verkürzt und die wiederholte Berechnung der Speicheradresse der indizierten Komponente a[i] vermieden.

In einem weiteren Beispiel nehmen wir an (vielleicht um sie schneller zu finden), dass gewisse Personengruppen im Array a miteinander verbunden sind. Die Information zur Verknüpfung wird in der Record-Struktur "Person" dargestellt durch eine zusätzliche Komponente "link". Damit werden die Records zu einer linearen Kette

verknüpft, so dass der Nachfolger und der Vorgänger jeder Person leicht gefunden werden können. Interessant an dieser Verknüpfungstechnik ist die Möglichkeit, die Kette in beiden Richtungen zu durchqueren mittels einer einzigen, in jedem Record gespeicherten Zahl. Die Technik funktioniert folgendermassen:

Wir nehmen an, dass die Indizes der drei aufeinanderfolgenden Kettenglieder i[k-1], i[k], i[k+1] sind. Der link-Wert zur Verknüpfung des k-ten Glieds sei i[k+1]-i[k-1]. Beim Durchlaufen der Kette in Vorwärtsrichtung wird i[k+1] aus den beiden aktuellen Index-Variablen x = i[k-1] und y = i[k] bestimmt zu

$$i[k+1] = x + a[y].link ,$$

während beim Durchqueren der Kette in Rückwärtsrichtung i[k-1] aus

	Vorname	Geschl.	Link
1	Carolyn	w	2
2	Chris	m	2
3	Tina	w	5
4	Robert	m	3
5	Jonathan	m	3
6	Karin	w	5
7	Robert	m	5
8	Marie	w	3
9	Anna	w	1
10	Mathias	m	3

Tabelle 1.2

x = i[k+1] und y = i[k] bestimmt wird zu

$$i[k-1] = x - a[y].link .$$

Ein Beispiel ist die Verknüpfung aller Personen gleichen Geschlechts in einer Tabelle (vgl. Tabelle 1.2).

Die Record- und die Array-Struktur haben die gemeinsame Eigen-

schaft, dass beide "willkürlich zugreifbare" Strukturen sind. Der Record ist insofern allgemeiner, als nicht alle Komponententypen identisch sein müssen. Der Array seinerseits bietet grössere Flexibilität, da die Komponenten-Selektoren berechenbare Werte sein können (dargestellt durch Ausdrücke), während die Selektoren von Record-Komponenten festgelegte, in der Record-Typdefinition enthaltene Namen sind.

1.8. Die Strukturart des varianten Record

In der Praxis ist es oft zweckmässig und natürlich, zwei Typen einfach als Varianten des gleichen Typs zu betrachten. Zum Beispiel kann der Typ Koordinate des vorangehenden Abschnitts für die Vereinigung der beiden Varianten der kartesischen und polaren Koordinaten gehalten werden, deren Komponenten zwei Längen, bzw. eine Länge und ein Winkel sind. Um die durch eine Variable tatsächlich angenommene Variante zu identifizieren, wird eine dritte Komponente eingeführt. Sie heisst Typ-Diskriminator oder "tag field".

```
type Koordinate =
     record case art: (kartesisch,polar) of
                      kartesisch: (x,y: real);
                      polar: (r: real; phi: Winkel)
     end
```

In diesem Beispiel ist der Name des Typ-Diskriminators art, und die Namen der Koordinaten sind x und y im Fall eines kartesischen Wertes oder r und phi im Fall eines polaren Wertes.

Der durch diesen Typ Koordinate bezeichnete Wertebereich ist die Vereinigung der zwei Typen

```
T1 = (x, y: real)
T2 = (r: real; phi: Winkel) .
```

Seine Kardinalität ist die Summe der Kardinalitäten von T1 und T2.

Sehr oft müssen nicht zwei völlig verschiedene Typen vereinigt werden, sondern zwei Typen mit teilweise identischen Komponenten. Wegen dieser vorherrschenden Situation wurde der Ausdruck variante Record-Struktur gebildet. Ein Beispiel hierfür ist der im voranstehenden Abschnitt definierte Typ Person, bei dem die wichtigen, in einem File zu registrierenden Eigenschaften vom Geschlecht einer Person abhängen. Zum Beispiel kann es bei einem Mann in einem besonderen Fall wichtig sein, welches Gewicht er hat und ob er einen Bart trägt, während bei einer Frau die drei Körpermasse als wissenswert angesehen werden (ihr Gewicht dagegen als vertraulich). Eine Typdefinition, die sich aus solchen Betrachtungen heraus ergibt, kann folgendermassen aussehen:

```
type Person=
     record name, vorname: alfa;
            geburtsdatum: Datum;
            zivilstand: (ledig,verheiratet,verwitwet,
                         geschieden);
          case geschlecht: (männlich,weiblich) of
          männlich: (gewicht: real; bärtig: Boolean);
          weiblich: (umfang: array [1..3] of integer)
     end
```

Die allgemeine Form einer varianten Record-Typdefinition ist

```
type T =                                                    (1.24)
     record s1: T1; ... ; sn: Tn;
          case sc: Tc of
          c1: (s11:T11; ... ; s1n1:T1n1);
                    ..................
          cm: (sm1:Tm1; ... ; smnm:Tmnm)
     end
```

Die si und sij sind die Selektor-Namen der Komponenten mit den Typen Ti und Tij, und sc ist der Name des Typ-Diskriminators vom Typ Tc. Die Konstanten c1...cm bezeichnen die Werte des (skalaren) Typs Tc. Eine Variable x vom Typ T besteht aus den Komponenten

x.s1, x.s2, ... , x.sn, x.sc, x.sk1, ... , x.sknk

genau dann, wenn der aktuelle Wert von x.sc gleich ck ist (x.sn = ck). Die Komponenten x.s1,...,x.sn bilden den gemeinsamen Teil der m Varianten.

Daraus folgt, dass die Verwendung eines Komponenten-Selektors x.skh (1 <= h <= nk) im Fall von x.sc <> ck als ein schwerer Programmierfehler betrachtet werden muss und (bezüglich des oben definierten Typs Person) zu der Frage führt, ob eine Dame einen Bart hat oder (im Fall von selektiver Aenderung) ihr ein Bart angehängt wird.

Beim Umgang mit varianten Records ist deshalb äusserste Vorsicht geboten, und entsprechende Operationen mit den einzelnen Varianten werden am besten in eine selektive Anweisung gruppiert, die sogenannte case-Anweisung, deren Struktur die der varianten Record-Typdefinition wiedergibt.

```
case x.sc of                                        (1.25)
  c1: S1;
  c2: S2;
   .....
  cm: Sm
end
```

Sk steht für die Anweisung, die den Fall behandelt, in dem x die Form von Variante k annimmt, d.h. sie wird genau dann zur Ausführung kommen, wenn der Diskriminator x.sc den Wert ck hat. Somit ist es ziemlich einfach, den Missbrauch von Selektor-Namen zu verhindern, indem geprüft wird, ob jedes Sk nur Selektoren

```
x.s1 ... x.sn       und
x.sk1 ... x.sknk    enthält.
```

Das folgende kurze Programmstück soll die Distanz zwischen zwei Punkten A und B berechnen, die durch die Variablen a und b vom varianten Record-Typ Koordinate gegeben sind. Der Rechenprozess ist je nach den vier möglichen Kombinationen der kartesischen und polaren Koordinaten verschieden (siehe Fig. 1.4).

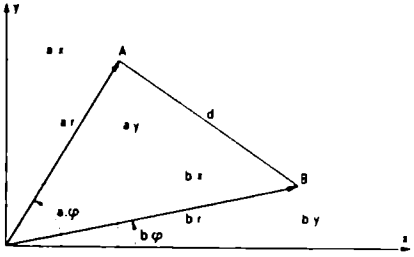


Fig. 1.4: Kartesische und Polarkoordinaten

```
case a.art of
   kartesisch: case b.art of
               kartesisch: d := sqrt(sqr(a.x-b.x)+sqr(a.y-b.y));
               polar:      d := sqrt(sqr(a.x-b.r*cos(b.phi)+
                                     sqr(a.y-b.r*sin(b.phi))
               end;
   polar:      case b.art of
               kartesisch: d := sqrt(sqr(a.r*sin(a.phi)-b.x)+
                                     sqr(a.r*cos(a.phi)-b.y));
               polar:      d := sqrt(sqr(a.r)+sqr(b.r)-
                                     2*a.r*b.r*cos(a.phi-b.phi))
               end
end
```

1.9. Die Strukturart "Menge" (Set)

Die dritte fundamentale Datenstruktur - neben Array und Record - ist die Strukturart Menge. Sie wird nach folgendem Schema definiert

```
type T = set of T0                                    (1.26)
```

Die möglichen Werte einer Variablen x vom Typ T sind Mengen von Elementen von T0. Die Menge aller Untermengen von Elementen einer Menge T0 wird die Potenzmenge von T0 genannt. Der Typ T umfasst somit die Potenzmenge seines Grundtyps T0.

Beispiele:

```
type intmenge = set of 0..30
type zeichenmenge = set of char
type bandzustand = set of Zustand
```

Das zweite Beispiel basiert auf der Standardmenge von Zeichen, die zum Typ char gehören; das dritte Beispiel auf einer Menge von Zuständen eines Magnetbandgerätes, die als ein skalarer Typ definiert sind (siehe Kapitel 1.3).

Gegeben seien die Variablen

```
im : intmenge
zm : zeichenmenge
t  : array [1..6] of bandzustand
```

Bestimmte Mengenwerte können durch Mengen-Konstruktoren gebildet und diesen Mengen-Variablen zugewiesen werden, wie folgendes Beispiel zeigt (im Gegensatz zur üblichen Notation verwenden wir für Mengen eckige statt geschweifte Klammern; letztere sind für Kommentare in Programmen reserviert):

```
im := [1,4,9,16,25]
zm := ['+','-','*','/']
t[3] := [manuell]
t[5] := []
t[6] := [ungeladen..schräglaufend]
```

Hierbei ist der t[3] zugewiesene Wert die einelementige Menge, die nur aus dem Element manuell besteht; t[5] wird die leere Menge zugewiesen, was bedeutet, dass sich das fünfte Band wieder im normalen (nicht Ausnahme-) Zustand befindet, während t[6] die Menge aller vier Ausnahmen zugeordnet wird.

Die Kardinalität eines Mengentyps T ist

```
Kardinalität(T) = 2**Kardinalität(T0)                    (1.27)
```

Dies ist leicht aus der Tatsache abzuleiten, dass jedes der Kardinalität(T0) Elemente von T0 durch einen der beiden Werte "anwesend"

und "abwesend" dargestellt werden muss und dass alle Elemente voneinander unabhängig sind. Für eine effiziente und wirtschaftliche Implementation ist es natürlich wichtig, dass der Grundtyp nicht nur endlich, sondern dass seine Kardinalität ziemlich klein ist.

Folgende Grundoperatoren sind für alle Mengentypen definiert:

* Mengen-Durchschnitt
\+ Mengen-Vereinigung
\- Mengen-Differenz
in Mengen-Einschluss

Das Bilden von Durchschnitt oder Vereinigung zweier Mengen heisst oft Mengenmultiplikation bzw. Mengenaddition; die Prioritäten der Mengenoperatoren werden entsprechend definiert, wobei der Durchschnittsoperator Priorität über Vereinigungs- und Differenzoperator hat, diese ihrerseits haben Vorrang über den Operator "Element von", der wie ein Relationsoperator eingestuft ist. Die folgenden Beispiele zeigen Mengenausdrücke und ihre vollständig geklammerten Aequivalenten:

```
r * s + t   =  (r*s) + t
r - s * t   =  r - (s*t)
r - s + t   =  (r-s) + t
x in s + t  =  x in (s+t)
```

Das erste Beispiel einer Anwendung der Mengenstruktur sei das Programm eines einfachen "Scanner" für einen Compiler. Wir nehmen an, dass es Aufgabe des Scanner ist, eine Folge von Zeichen in eine Folge von Texteinheiten der zu compilierenden Sprache zu übersetzen, in sogenannte Sprach-Symbole. Der Scanner soll als eine Prozedur dargestellt werden, die bei jedem Aufruf hinreichend viele Eingabe-Zeichen liest, um genau ein nächstes Ausgabe-Symbol zu generieren. Dabei sollen folgende Uebersetzungsregeln gelten:

1. Die Menge der Ausgabe-Symbole besteht aus den Elementen Bezeichner, Zahl, kleinergleich, grössergleich, wird und anderen, die den Zeichen +, -, *, usw. entsprechen.

2. Das Symbol Bezeichner wird beim Lesen einer mit einem Buchstaben

beginnenden Folge von Buchstaben und Ziffern generiert.

3. Das Symbol Zahl wird beim Lesen einer Folge von Ziffern erzeugt.

4. Die Symbole kleinergleich, grösseroleich und wird werden beim Lesen der jeweiligen Zeichenpaare <=, >=, := generiert.

5. Leerzeichen und Zeilenenden werden übersprungen.

Wir haben eine einfache Prozedur read(x) zur Verfügung, die das nächste Zeichen aus der Eingabe-Sequenz holt und es der Variablen x zuweist. Das resultierende Ausgabe-Symbol soll einer globalen Variablen mit Namen sym zugewiesen werden. Ausserdem existieren globale Variable id und num, deren Bedeutung sich aus Programm 1.2 ergibt. Eine weitere Variable ch enthält das gerade untersuchte Zeichen der Eingabe-Sequenz.

```
var ch: char;
    sym: symbol;
    num: integer;
    id: record
          k: 0 .. maxk;
          a: array [1..maxk] of char
        end;

procedure scanner;
   var ch1: char;
begin {überspringe Leerzeichen}
   while ch = ' ' do read(ch);
   if ch in ['a'..'z'] then
      with id do
      begin sym := identifier;  k := 0;
         repeat if k < maxk then
                   begin k := k+1; a[k] := ch
                   end;
                   read(ch)
         until not (ch in ['a'..'z', '0'..'9'])
      end else
   if ch in ['0'..'9'] then
      begin sym := number;  num := 0;
```

```
            repeat num := 10*num + ord(ch)-ord('0');
               read(ch)
            until not (ch in ['0'..'9'])
         end else
      if ch in ['<',':','>'] then
         begin ch1 := ch; read(ch);
            if ch = '=' then
            begin
               if ch1 = '<' then sym := kgl else
               if ch1 = '>' then sym := ggl else sym := wird;
               read(ch)
            end
            else sym := s[ch1]
         end else
         begin {andere Symbole}
            sym := s[ch]; read(ch)
         end
   end {scanner}
```

Programm 1.2

S bezeichnet eine Abbildung von Zeichen in Symbole, d.h. einen Array von Symbolen mit einem Indexbereich über den Zeichen, die weder Ziffern noch Buchstaben sind. Der Gebrauch einer Menge von Zeichen zeigt, wie ein Scanner, unabhängig von der Reihenfolge der Zeichen, in der zugrunde liegenden Zeichenmenge programmiert werden kann.

Ein zweites Beispiel behandelt das Erstellen eines Stundenplanes. Nehmen wir an, dass M Studenten ihre Wahl zwischen N Fächern getroffen haben. Nun soll ein Stundenplan so aufgestellt werden, dass gewisse Fächer parallel unterrichtet werden, ohne dass Konflikte entstehen [13].

Im allgemeinen ist das Erstellen eines Stundenplans ein äusserst schwieriges kombinatorisches Problem, das unter Berücksichtigung vieler Nebenbedingungen und Faktoren zu lösen ist. In diesem Beispiel wollen wir das Problem drastisch vereinfachen, ohne zu behaupten, eine Lösung für eine realistische Stundenplansituation anzugeben.

Wir wollen zuerst eine geeignete Wahl für die Fächer treffen, die parallel unterrichtet werden können. Dabei können wir unsere Entscheidungen auf eine Menge von Daten stützen, die aus den Einschreibungen der einzelnen Studenten ableitbar sind, nämlich aus der Aufzählung der Fächer, die nicht gleichzeitig gehalten werden können. Daher programmieren wir zuerst den Prozess der Datenreduktion, der von folgenden Vereinbarungen ausgeht und der annimmt, dass die Studenten von 1 bis M und die Kurse von 1 bis N durchnumeriert sind.

```
type kurs = 1..N;                                          (1.28)
     student = 1..M;
     auswahl = set of kurs;

var s: kurs;
    i: student;
    einschreibung: array [student] of auswahl;
    konflikt: array [kurs] of auswahl;

{Bestimme die Menge der Kurse, die mit den von einem Studenten
 gewählten Kursen in Konflikt stehen}

for s := 1 to N do  konflikt[s] := [];
for i := 1 to M do

      if s in einschreibung[i] then
         konflikt[s] := konflikt[s] + einschreibung[i]
```

(Beachte, dass s in konflikt[s] eine Folge dieses Algorithmus ist.)

Die Hauptaufgabe besteht nun im Aufstellen des Stundenplans, d.h. einer Liste von Fächergruppen (session), die jeweils Kurse umfassen, die keine Konflikte verursachen. Aus der ganzen Menge der Kurse wählen wir geeignete, konfliktfreie Teilmengen von Kursen, subtrahieren sie so lange von einer Variablen namens restliche, bis die Menge der restlichen Kurse leer ist.

```
var k: integer;                                            (1.29)
    restliche,session: auswahl;
    stundenplan: array [1..N] of auswahl;
```

```
k := 0; restliche := [1..N];
while restliche <> [] do
   begin session := "geeignete nächste Wahl";
         restliche := restliche - session;
         k := k+1; stundenplan[k] := session
   end
```

Wie treffen wir nun eine "geeignete nächste Wahl"? Am Anfang können wir irgendeinen Kurs aus der Menge der restlichen Kurse wählen. Die Wahl weiterer Kandidaten muss dann auf die Menge der Kurse beschränkt werden, die Elemente der Menge der restlichen Kurse sind und nicht mit den zuerst gewählten kollidieren. Wir nennen diese Menge die Testmenge. Wenn wir einen Kandidaten aus der Testmenge untersuchen, hängt seine endgültige Wahl davon ab, ob der Durchschnitt der bereits gewählten Kurse mit der Menge der mit dem Kandidaten kollidierenden Kurse leer ist. Dies führt zur Ausarbeitung der Anweisung

```
session := "nächste geeignete Wahl"
```

```
var s,t: kurs;                                          (1.30)
    testmenge: auswahl;

begin s := 1;
      while not(s in restliche) do s := s+1;
      session := [s]; testmenge := restliche - konflikt[s];
      for t := 1 to N do
         if t in testmenge then
            begin if konflikt[t] * session = [] then
                     session := session + [t]
            end
end
```

Offensichtlich wird diese Lösung für die Wahl geeigneter Fächergruppen keinen Stundenplan erstellen, der in irgendeinem spezifischen Sinn optimal ist. In ungünstigen Fällen kann die Zahl der Fächergruppen gleich gross sein wie die Zahl der Kurse, obwohl gleichzeitiges Abhalten von Fächern möglich wäre.

Struktur	Array	Record	Set
Vereinbarung	a: array [I] of TØ	r: record s1: T1; ... sn: Tn end	s: set of TØ
Selektor	a[i] (i in I)	r.s (s in {s1...sn})	keinen
Zugriff zu Komponenten	Selektor mit berechenbarem Index I	Selektor mit vereinbartem Komponentennamen S	Test auf Enthaltensein mit in
Komponententypen	alle identisch (TØ)	können alle verschieden sein	alle identisch (TØ)
Kardinalität	card(TØ)**card(I)	card(T1) * ... * card(Tn)	2**card(TØ)

Tabelle 1.3

1.10. Darstellung von Array-, Record- und Mengenstrukturen

Das Wesentliche bei der Verwendung von Abstraktionen in der Programmierung ist, dass ein Programm allein aufgrund der für die Abstraktion geltenden Gesetze entwickelt, verstanden und verifiziert werden kann und dass dazu weitere Kenntnisse über die Art der Implementation und Darstellung der Abstraktionen in einem bestimmten Computer unnötig sind. Trotzdem ist es für einen erfolgreichen Programmierer hilfreich, wenn er die allgemein verwendeten Techniken für die Darstellung der Grundkonzepte von Programmierabstraktionen versteht, wie z.B. die der Datenstrukturen. Es ist insofern hilfreich, als es dem Programmierer erlaubt, vernünftige Entscheidungen über Programm- und Datenentwurf nicht nur im Licht der

abstrakten Eigenschaften der Strukturen, sondern auch im Hinblick auf ihre Darstellung in einer bestimmten Rechenanlage zu treffen und dabei ihre besonderen Vor- und Nachteile zu berücksichtigen.

Das Problem der Datendarstellung ist das der Abbildung der abstrakten Struktur auf einen Computerspeicher. Speicher sind - in einer ersten Näherung - Arrays von individuellen Speicherzellen, sogenannten Worten. Die Indizes dieser Worte heissen Adressen.

var speicher: array [Adresse] of Wort (1.31)

Die Kardinalität der Typen Adresse und Wort wechselt von einer Rechenanlage zur anderen. Ein spezielles Problem ist die grosse Verschiedenartigkeit der Kardinalität des Typs Wort. Der Logarithmus der Kardinalität heisst Wortgrösse, da er die Zahl der Bits angibt, die eine Speicherzelle umfasst.

1.10.1. Darstellung von Arrays

Eine Darstellung einer Array-Struktur ist eine Abbildung des (abstrakten) Array mit Komponenten vom Typ T auf einen Speicher, der selbst ein Array mit Komponenten vom Typ Wort ist.

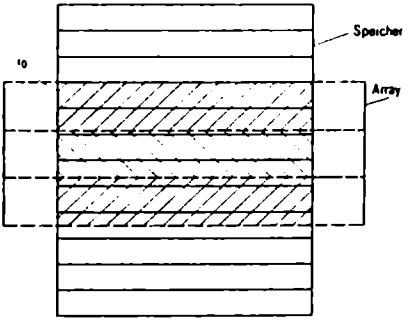


Fig. 1.5: Abbildung eines Array im Speicher

Der Array sollte so abgebildet werden, dass die Berechnung der Adressen von Array-Komponenten so einfach (und deshalb so effizient) wie möglich ist. Die Adresse oder der Speicherindex i

der j-ten Array-Komponente wird berechnet durch die lineare Abbildungsfunktion

$$i = i0 + j*s \quad , \tag{1.32}$$

wobei i0 die Adresse der ersten Komponente ist und s die Zahl der Worte angibt, die eine Komponente "belegt". Da das Wort nach Definition die kleinste individuell zugreifbare Speichereinheit bildet, ist es höchst wünschenswert, dass s eine ganze Zahl ist: der einfachste Fall ist s = 1. Ist s keine ganze Zahl (und das ist der Normalfall), dann wird s gewöhnlich auf die nächste ganze Zahl aufgerundet. Jede Array-Komponente besetzt dann ganz(s) Worte, wobei ganz(s)-s Worte unbenutzt bleiben (vgl. Fig. 1.5 und 1.6.).

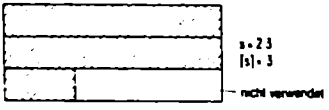


Fig. 1.6: Abbildung eines Record

Das Aufrunden der benötigten Anzahl Worte auf die nächste ganze Zahl heisst Auffüllen (padding). Der Speicherausnutzungsfaktor u ist der Quotient des minimal zur Darstellung einer Struktur benötigten Speichers s und des wirklich benötigten Speichers s':

$$u = s : s' = s : ganz(s) \tag{1.33}$$

Da ein Implementator eine Speicherausnutzung möglichst nahe bei 1 zu erreichen sucht und da der Zugriff zu Teilen eines Wortes umständlich und relativ ineffizient ist, wird er einen Kompromiss zu schliessen haben. Dabei sind folgende Punkte zu berücksichtigen:

1. Auffüllen verkleinert die Speicherausnutzung.

2. Verzicht auf Auffüllen kann ineffizienten Zugriff zu Wortteilen erfordern.

3. Durch Zugriff zu Wortteilen kann der Code (übersetztes Programm) grösser und deshalb der durch Verzicht auf Auffüllen erzielte

Gewinn verkleinert werden.

Tatsächlich überwiegen die Punkte 2 und 3 so sehr, dass Compiler immer automatisch auffüllen. Wir bemerken, dass für den Ausnutzungsfaktor immer gilt: u > 0.5, wenn s > 0.5. Ist aber s <= 0.5, dann kann der Ausnutzungsfaktor wesentlich verbessert werden, wenn man mehr als eine Array-Komponente in jedes Wort setzt. Diese Technik heisst Packen. Werden n Komponenten in ein Wort gepackt, so ergibt sich der Ausnutzungsfaktor zu

$$u = n*s \; / \; \text{ganz}(n*s) \qquad (1.34)$$

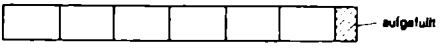


Fig. 1.7: Packen von 6 Komponenten in ein Wort

Zugriff zu der i-ten Komponente eines gepackten Array bedingt die Berechnung der Wortadresse j, in der die gewünschte Komponente sitzt, und die Berechnung der Komponentenposition k innerhalb des Wortes.

$$j = i \; \underline{\text{div}} \; n \qquad (1.35)$$
$$k = i \; \underline{\text{mod}} \; n = i - j*n$$

In den meisten Programmiersprachen hat der Programmierer keine Kontrolle über die Darstellung der abstrakten Strukturen. Es sollte aber möglich sein anzugeben, wenn Packen gewünscht wird, zumindest in den Fällen, in denen mehr als eine Komponente in ein einzelnes Wort passt, d.h. wenn der Speichergewinn einen Faktor 2 und mehr beträgt. Wir führen die Konvention ein, den Wunsch nach Packen durch Voranstellen des Symbols packed vor das Symbol array (oder record) in der Deklaration anzuzeigen.

Beispiel:

```
type alfa = packed array [1..n] of char
```

Diese Möglichkeit ist besonders nützlich bei Rechnern mit grosser Wortlänge und relativ leichtem Zugriff zu Teilfeldern eines Wortes. Die wesentliche Eigenschaft dieses Zusatzes ist, dass er die

Bedeutung (oder Korrektheit) eines Programms in keiner Weise ändert. Das bedeutet, dass leicht eine andere Darstellung gewählt werden kann mit der impliziten Garantie, dass die Bedeutung des Programms unverändert bleibt.

Die Kosten für den Zugriff zu Komponenten eines gepackten Array können in vielen Fällen wesentlich reduziert werden, wenn der ganze Array auf einmal ausgepackt (oder gepackt) wird. Das kommt daher, dass ein effizienter sequentieller Durchlauf durch den ganzen Array möglich ist und somit unnötiges Auswerten einer komplizierten Abbildungsfunktion für jede individuelle Komponente entfällt. Wir postulieren deshalb die Existenz zweier Standard-Prozeduren pack und unpack, wie sie unten definiert werden. Gegeben seien die Variablen

```
u : array [a..d] of T
p : packed array [b..c] of T
```

wobei $c-d < d-a$ sind. Dann ist

```
pack(u,i,p), (a <= i <= b-c+d)                    (1.36)
```

äquivalent zu

```
p[j] := u[j+i-b],        j = b...c
```

und

```
unpack(p,i,u), (a <= i <= b-c+d)                  (1.37)
```

äquivalent zu

```
u[j+i-b] := p[j],        j = b...c.
```

1.10.2. Darstellung von Records

Records werden auf den Computerspeicher durch einfaches Nebeneinandersetzen ihrer Komponenten abgebildet. Die Adresse der i-ten Komponente (eines Feldes) relativ zur Anfangsadresse des Record r heisst Offset k[i] der Komponente. Er berechnet sich zu

```
k[1] = s[1] + s[2] + ... + s[i-1]                 (1.38)
```

wobei s[j] die Grösse (in Worten) der j-ten Komponente ist. Die Tatsache, dass alle Komponenten eines Array vom gleichen Typ sind, hat zur Folge, dass

$$s[1] = s[2] = \ldots = s[n] = s$$

ist und deshalb

$$k[i] = s[1] + \ldots + s[i-1] = (i-1)*s$$

ist. Die Allgemeinheit der Record-Struktur erlaubt normalerweise keine solch einfache lineare Funktion zur Berechnung der Offset-Adresse und ist deshalb der eigentliche Grund dafür, dass Record-Komponenten nur über feste Namen zugegriffen werden können. Diese Einschränkung hat die angenehme Konsequenz, dass die jeweiligen Offsets zur Compilationszeit bekannt sind. Die sich daraus ergebende grössere Effizienz des Zugriffs zum Record-Feld ist bekannt.

Das Problem des Packens kommt auf, wenn mehrere Record-Komponenten wie in Fig. 1.8 in ein einzelnes Speicherwort passen. Wir wollen wieder annehmen, dass Voranstellen des Symbols packed vor das Symbol record in der Deklaration den Wunsch nach Packen angibt. Da Offsets von einem Compiler berechnet werden können, kann auch der Offset einer Komponente in einem Wort vom Compiler bestimmt werden. Das bedeutet, dass auf vielen Rechenanlagen das Packen von Records eine wesentlich kleinere Minderung der Effizienz beim Zugriff verursacht als das Packen von Arrays.

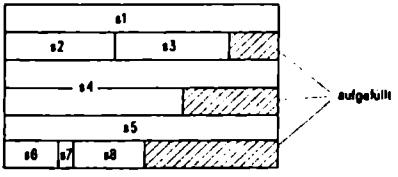


Fig. 1.8 Darstellung eines gepackten Record

1.10.3. Darstellung von Mengen

Eine Menge m ist in einem Computerspeicher einfach dargestellt durch ihre charakteristische Funktion C(m). Dies ist ein Array von logischen Werten, dessen i-te Komponente die An- oder Abwesenheit des Wertes i in der Menge angibt.

$$C(m)i = (i \text{ in } m) \qquad (1.39)$$

Zum Beispiel lässt sich die Menge von kleinen Zahlen

$$m = [1,4,8,9]$$

darstellen durch die Folge der logischen Werte F (false) und T (true)

$$C(m) = (FTFFTFFFTT),$$

wenn der zugrunde liegende Typ von m der ganzzahlige Unterbereich 0..9 ist. In einem Computerspeicher ist die Folge der logischen Werte durch eine sogenannte Bit-Folge dargestellt (vgl. Fig. 1.9).

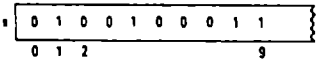


Fig. 1.9: Darstellung einer Mengenvariablen durch die charakteristische Funktion

Die Darstellung von Mengen durch ihre charakteristische Funktion hat den Vorteil, dass die Operationen zur Berechnung der Vereinigung, des Durchschnitts und der Differenz zweier Mengen so einfach implementiert werden können wie elementare logische Operationen. Die folgenden Aequivalenzen, die für alle Elemente i des Grundtyps der Mengen x und y gelten, setzen die logischen Operationen mit Mengenoperationen in Relation:

$$\begin{aligned} &i \text{ in } (x+y) = (i \text{ in } x) \text{ or } (i \text{ in } y) \\ &i \text{ in } (x*y) = (i \text{ in } x) \text{ and } (i \text{ in } y) \\ &i \text{ in } (x-y) = (i \text{ in } x) \text{ and not}(i \text{ in } y) \end{aligned} \qquad (1.40)$$

Diese logischen Operationen sind auf allen digitalen Rechenanlagen verfügbar und arbeiten sogar gleichzeitig auf allen entsprechenden Elementen (Bits) eines Wortes. Um grundlegende Mengenoperationen effizient implementieren zu können, scheint es daher richtig, Mengen in einer kleinen festen Anzahl von Worten darzustellen, auf der nicht nur die elementaren logischen Operationen, sondern auch die Shift-Operationen verfügbar sind. Test auf Vorhandensein eines Elements in einer Menge wird dann implementiert durch eine einzige Shift-Operation und einen anschliessenden (Vorzeichen) Bit-Test. Damit kann z.B. ein Test der Form

x in [c1, c2, ... , cn]

wesentlich effizienter implementiert werden als der übliche äquivalente Boolesche Ausdruck

(x = c1) or (x = c2) or ... or (x = cn)

Als Koroller ergibt sich, dass die Struktur der Menge nur für Grundtypen kleiner Kardinalität verwendet werden soll. Die Kardinalität der Grundtypen, für die eine effiziente Implementation garantiert werden kann, wird durch die Wortlänge des jeweiligen Computers begrenzt. Selbstverständlich sind Computer mit grossen Wortlängen in dieser Beziehung bevorzugt. Bei realtiv kleiner Wortlänge ist eine Darstellung zu wählen, die eine kleine Zahl von Worten für eine Menge verwendet.

1.11. Die Struktur des sequentiellen Files

Eine gemeinsame Charakteristik der bisher vorgestellten Datenstrukturen, nämlich der Array-, der Record- und der Set-Struktur, ist ihre endliche Kardinalität (vorausgesetzt, dass die Kardinalität ihrer Komponenten-Typen endlich ist). Deshalb bieten sie wenig Schwierigkeit für den Implementator; geeignete Darstellungen lassen sich auf einer digitalen Rechenanlage leicht finden.

Die sogenannten nichtelementaren Strukturen - Sequenzen, Bäume,

Graphen, usw. - sind charakterisiert durch ihre nicht endliche Kardinalität. Dieser Unterschied zu den grundlegenden Strukturen mit endlicher Kardinalität ist von entscheidender Bedeutung und hat wesentliche praktische Konsequenzen. Betrachten wir z.B. die Definition der Sequenz-Struktur:

> Eine Sequenz mit Grundtyp T0 ist entweder die leere Sequenz oder die Verkettung einer Sequenz (mit Grundtyp T0) mit einem Wert vom Typ T0.

Der so definierte Sequenz-Typ T umfasst offensichtlich unendlich viele Werte. Jeder Wert selbst besteht aus einer endlichen Zahl von Komponenten vom Typ T0, aber diese Zahl ist unbeschränkt, d.h. zu jeder solchen Sequenz kann eine längere konstruiert werden.

Analoge Betrachtungen können auf alle anderen "nichtelementaren" Datenstrukturen angewandt werden. Die erste Folge ist, dass der zur Darstellung eines Wertes eines nichtelementaren strukturierten Typs notwendige Speicherplatz zur Compilationszeit nicht bekannt ist; ja, er mag sogar während der Ausführung des Programms variieren. Das verlangt ein Verfahren zur dynamischen Speicherplatzzuweisung, bei dem Speicher belegt wird, wenn die zugehörigen Werte "wachsen", und bei dem Speicher möglicherweise für andere Verwendungen wieder freigegeben wird, wenn die Werte "schrumpfen". Es ist daher klar, dass die Frage der Darstellung von nichtelementaren Strukturen tückisch und schwierig ist und dass ihre Lösung die Effizienz eines Prozesses und die Wirtschaftlichkeit der Verwendung des Speichers wesentlich beeinflussen. Eine geeignete Wahl kann nur getroffen werden aufgrund von Kenntnissen über die primitiven Operationen, die auf der Struktur ausgeführt werden sollen, und über die Häufigkeit ihrer Ausführung. Da dem Designer einer Sprache und des dazugehörigen Compilers diese Information nicht zur Verfügung steht, ist er wohlberaten, wenn er nichtelementare Strukturen aus einer (allgemein verwendbaren) Sprache ausklammert. Dementsprechend wird dem Programmierer empfohlen, nichtelementare Strukturen zu meiden, wenn immer das Problem unter Verwendung elementarer Strukturen behandelt werden kann.

Das Dilemma, nichtelementare Datenstrukturen zur Verfügung stellen zu müssen, ohne Information über ihren möglichen Gebrauch zu haben,

wird in den meisten Sprachen und Compilern umgangen unter Verwendung der Erkenntnis, dass sich alle diese Strukturen letztlich entweder aus unstrukturierten Elementen oder fundamentalen Strukturen zusammensetzen. Beliebige Strukturen können daher durch explizite, vom Programmierer spezifizierte Operationen generiert werden, wenn die dynamische Zuweisung und Verknüpfung der Komponenten explizit formulierbar gemacht wird. Techniken zur Erstellung und Veränderung dieser nichtelementaren Strukturen werden in Kapitel 4 behandelt.

Es existiert aber eine Struktur, die in bezug auf ihre unendliche Kardinalität nicht elementar ist, die aber so allgemein und häufig verwendet wird, dass sie in die Menge der grundlegenden Strukturen einbezogen werden sollte: die Sequenz. Um den abstrakten Begriff der Sequenz zu beschreiben, wird folgende Terminologie eingeführt:

1. < > beschreibt die leere Sequenz.

2. <x0> bezeichnet die aus der einen Komponente x0 bestehende Sequenz; sie heisst auch einelementige Sequenz.

3. Wenn x = <x1,...,xm> und y = <y1,...,yn> Sequenzen sind, so ist

$$x \,\&\, y = \langle x1,\ldots,xm,y1,\ldots,yn\rangle \qquad (1.41)$$

   die Verkettung von x und y.

4. Bei einer nicht leeren Sequenz x = <x1,...,xn> bezeichnet

$$first(x) = x1 \qquad (1.42)$$

   das erste Element von x.

5. Bei einer nicht leeren Sequenz x = <x1,...,xn> bezeichnet

$$rest(x) = \langle x2,\ldots xn\rangle \qquad (1.43)$$

   die Sequenz x ohne ihr erstes Element. Folglich gilt die invariante Relation

$$\langle first(x)\rangle \,\&\, rest(x) = x \qquad (1.44)$$

Die Einführung dieser Notation bedeutet nicht, dass sie in wirklichen Programmen zur Ausführung durch Computer verwendet wird. Vielmehr ist es wesentlich, die Verkettungsoperation nicht in ihrer Allgemeinheit zu verwenden und die Behandlung von Sequenzen auf die Anwendung einer sorgfältig ausgewählten Menge von Operatoren zu beschränken; diese Operatoren, die selbst anhand der abstrakten Begriffe der Sequenz und der Verkettung definiert sind, sollen eine gewisse Disziplin bei ihrer Anwendung garantieren. Eine sorgfältige Wahl der Menge der Sequenz-Operatoren ermöglicht dem Implementator, eine geeignete und effiziente Darstellung der Sequenz auf irgendeinem gegebenen Speichermedium zu wählen, und stellt sicher, dass der zugehörige Mechanismus für die dynamische Speicherzuweisung hinreichend einfach sein kann, damit der Programmierer dessen technische Einzelheiten nicht beachten muss.

Um zum Ausdruck zu bringen, dass die als grundlegende Datenstruktur einzuführende Sequenz nur die Anwendung einer eingeschränkten Menge von Operatoren zulässt, die im wesentlichen nur den streng sequentiellen Zugriff zu den Komponenten erlauben, heisst diese Struktur sequentielles File oder einfach File. In enger Analogie zur Notation für Array- und Mengen-Definitionen wird ein File-Typ definiert durch:

```
type T = file of T0                                    (1.45)
```

Diese Formel besagt, dass irgendein File vom Typ T aus null oder mehr Komponenten vom Typ T0 besteht.

Beispiel:

```
type text = file of char
type stapel = file of Karte
```

Das Wesentliche des sequentiellen Zugriffs ist, dass zu jeder Zeit nur zu einer einzelnen spezifischen Komponente der Sequenz unmittelbar zugegriffen werden kann. Diese Komponente ist spezifiziert durch die gegenwärtige Position des Zugriffsmechanismus. Die Position kann durch File-Operatoren geändert werden, normalerweise entweder zur nächsten Komponente oder zum ersten Element der ganzen Sequenz. Wir drücken die File-Position formal dadurch aus, dass wir

ein File x als aus zwei Teilen bestehend betrachten, aus einem Teil xL links von der gegenwärtigen Position und aus einem Teil xR rechts davon. Offensichtlich drückt die Gleichung

$$x = xL \ \& \ xR \qquad (1.46)$$

eine invariante Relation aus.

Eine zweite wesentliche Konsequenz des sequentiellen Zugriffs ist, dass die Prozesse des Erstellens und Durchsuchens einer Sequenz strikte getrennt sind und nicht in beliebiger Reihenfolge gemischt werden können. Ein File wird durch wiederholtes Anhängen von Komponenten (am Ende) erstellt und kann dann mit sequentiellem Durchlauf gelesen werden. Es ist daher üblich, einem File zwei Zustände zuzuordnen: den Zustand des Erstelltwerdens (schreiben) oder den des Durchsuchtwerdens (lesen).

Der Vorteil des streng sequentiellen Zugriffs kommt besonders zur Geltung, wenn Files auf einem Hintergrundspeicher gespeichert sind, d.h. wenn Uebertragungen zwischen verschiedenen Speichermedien vorkommen. Der sequentielle Zugriff ist die einzige Möglichkeit, um die mit einer solchen Uebertragung verbundenen Schwierigkeiten vor dem Programmierer erfolgreich zu verbergen. Im besonderen erlaubt er die Anwendung einfacher Pufferungstechniken, die allein eine optimale Verwendung der verschiedenen, in einem komplexen Computer-System zur Verfügung stehenden Einheiten garantiert.

Es gibt gewisse Speichermedien, bei denen sequentieller Zugriff sogar der einzig mögliche ist. Darunter befinden sich sicherlich alle Arten von Bändern. Aber auch auf magnetischen Trommeln und Platten bildet jede Speicherspur eine Speichereinheit, die nur sequentiellen Zugriff erlaubt. Streng sequentieller Zugriff ist die primäre Charakteristik jeder mechanisch bewegten Speichereinheit und sogar auch einiger anderer.

1.11.1. Elementare File-Operatoren

Wir wollen nun weiter den abstrakten Begriff des sequentiellen File-Zugriffs formulieren durch einen Satz konkreter elementarer File-Operatoren, die dem Programmierer zur Verfügung stehen. Sie werden mit Hilfe der Begriffe der Sequenz und der Verkettung definiert. Es gibt einen Operator zum Eröffnen des Prozesses der File-Erstellung, einen zum Eröffnen der Inspektion, einen zum Anhängen einer Komponente an das Ende einer Sequenz und einen zum Weiterschreiten zur nächsten Komponente bei der Inspektion. Diese Operatoren verwenden in der hier angegebenen Form eine implizite Hilfsvariable, die ein Pufferelement darstellt. Wir nehmen an, dass ein solcher Puffer jeder File-Variablen x automatisch zugeordnet ist, und bezeichnen ihn mit $x\uparrow$. Wenn x vom Typ T ist, dann ist $x\uparrow$ vom zugehörigen Grundtyp T0.

1. Erstellen der leeren Sequenz. Die Operation

$$\text{rewrite}(x) \tag{1.47}$$

steht für die Zuweisung

$$x := \langle\,\rangle\ .$$

Die Operation überschreibt das gegenwärtige x und initialisiert den Prozess des Erstellens einer neuen Sequenz.

2. Ausdehnen einer Sequenz. Die Operation

$$\text{put}(x) \tag{1.48}$$

steht für die Zuweisung

$$x := x \,\&\, \langle x\uparrow\rangle\ ,$$

die effektiv den Wert $x\uparrow$ an die Folge x anhängt.

3. Initialisierung der Inspektion. Die Operation

$$\text{reset}(x) \tag{1.49}$$

steht für die gleichzeitigen Zuweisungen

$$xL := \langle\,\rangle$$
$$xR := x$$
$$x\uparrow := first(x)\ .$$

4. Fortschreiten zum nächsten Element. Die Operation

$$get(x) \qquad (1.50)$$

steht für die gleichzeitigen Zuweisungen

$$xL := xL \,\&\, \langle first(xR)\rangle$$
$$xR := rest(xR)$$
$$x\uparrow := first(rest(xR))$$

Man beachte, dass first(s) nur definiert ist, wenn s nicht leer ist.

Die Operatoren rewrite und reset hängen offensichtlich nicht von der Position ab, die das File vor ihrer Ausführung hat. Sie positionieren das File in jedem Fall an seinen Anfang.

Wenn man eine Sequenz durchläuft, muss man das Ende der Sequenz erkennen können, da sonst die Zuweisung

$$x\uparrow := first(xR)$$

eine undefinierte Operation darstellt. Das Erreichen des File-Endes ist sicherlich gleichbedeutend mit der Aussage, dass der rechte Teil xR leer ist. Wir führen deshalb das Prädikat ein

$$eof(x) = (xR = \langle\,\rangle) \qquad (1.51)$$

mit der Bedeutung, das Ende des Files (end of file) ist erreicht. Die Operation get(x) kann deshalb nur ausgeführt werden, wenn eof(x) den Wert false hat.

Im Prinzip ist es möglich, alle File-Operationen durch die vier File-Grundoperatoren auszudrücken. In der Praxis ist es aber oft

natürlich, die Operation des Weitersetzens der File-Position (get oder put) mit dem Zugriff zur Puffervariablen zu verknüpfen. Deshalb führen wir zwei weitere Prozeduren ein, die sich durch die Grundoperatoren beschreiben lassen. Sei v eine Variable und e ein Ausdruck vom Typ T0 der File-Komponenten. Dann steht

```
read(x,v)  für
     v := x↑; get(x)
```

und

```
write(x,e)  für
     x↑ := e; put(x)
```

Der Vorteil der Verwendung von read und write anstelle von get und put liegt nicht nur in der Kürze, sondern auch in der konzeptionellen Einfachheit. Man kann nun nämlich die Existenz der Puffervariablen x↑ ignorieren, deren Wert manchmal undefiniert ist. Die Puffervariable kann aber für das "Vorausschauen" [look ahead] nützlich sein.

Notwendig für die Ausführung der beiden Prozeduren sind die beiden Bedingungen

```
not eof(x)   für   read(x,v)
    eof(x)   für   write(x,e)
```

Beim Lesen erhält das Prädikat eof(x) den Wert true, sobald das letzte Element des Files gelesen wurde. Diese Betrachtungen sind in zwei Programmschemata für das sequentielle Erstellen und Verarbeiten eines Files x zusammengefasst. Die Anweisungen A, S und das Prädikat p sind zusätzliche Parameter dieser Schemata.

Schreiben eines Files x

```
rewrite(x);                                          (1.52)
while p do
   begin A(v); write(x,v)
   end
```

Lesen eines Files

```
reset(x);                                          (1.53)
while not eof(x) do
   begin read(x,v); S(v)
   end
```

1.11.2. Files mit Unterstrukturen

In der Mehrzahl der Anwendungen erfordern grosse Files irgendeine Art von Unterstruktur. Ein Buch z.B. kann zwar als eine einzige Folge von Zeichen aufgefasst werden, ist aber in Kapitel oder Paragraphen unterteilt. Die Unterstruktur soll explizite Anhaltspunkte, etwas wie Koordinaten zur Verfügung stellen, um die Orientierung in der langen Sequenz der Information zu erleichtern. Bestehende Speichereinheiten bieten oft die Möglichkeit, solche Markierungspunkte darzustellen (z.B. Bandmarken); das Aufsuchen solcher Marken ist wesentlich rascher als das Durchsuchen aller dazwischenliegenden Information.

Im Rahmen unserer Notation ist es natürlich, eine erste Stufe einer Unterstruktur dadurch einzuführen, dass man ein solches File als Sequenz von Komponenten betrachtet, die selbst wieder Sequenzen sind, d.h. als File von Files. Nehmen wir an, dass die eigentlichen Komponenten (oder Einheiten) vom Typ U sind, dann hat die Unterstruktur den Typ

```
T' = file of U
```

und das ganze File hat den Typ

```
T = file of T' .
```

Es ist klar, dass auf diese Art Files erstellt werden können mit einer Unterteilung bis zu einer beliebig tiefen Verschachtelung. Im allgemeinen kann ein Typ Tn definiert werden durch die rekursive Relation

```
T(i) = file of T(i-1)        i = 1...n
```

mit T0 = U. Solche Files heissen oft Mehrstufen-Files, und eine Komponente vom Typ Ti heisst Segment von der Stufe i. Ein Beispiel eines Mehrstufen-Files ist ein Buch, wobei die Stufen der Segmentierung den Kapiteln, Abschnitten, Paragraphen und Zeilen entsprechen. Der weitaus häufigste Fall aber ist das File mit einer Stufe von Segmentierung.

Dieses einstufig segmentierte File entspricht keineswegs einem Array von Files. Denn nach dem bisher Gesagten ist die Zahl der Segmente variabel, und das File kann am Ende immer noch erweitert werden. Bleiben wir im Rahmen unserer bisherigen Notation, und nehmen wir ein File, das durch

x: file of file of U

definiert ist, dann bezeichnet x↑ das gegenwärtig zugreifbare Segment und x↑↑ die gegenwärtig zugreifbare Grundkomponente. Entsprechend beziehen sich put(x↑↑) und get(x↑↑) auf eine Grundkomponente, während put(x↑) und get(x↑) die Operationen des Anhängens und Weiterschreitens zum nächsten Segment bezeichnen.

Segmentierte Files sind auf praktisch allen sequentiellen Speichereinheiten inklusive Bändern einfach zu implementieren. Ihre Segmentierung ändert die wesentliche Charakteristik nicht, die nur sequentiellen Zugriff zu einer einzelnen Komponente oder - durch einen möglicherweise schnelleren Positionierungsmechanismus - zu Segmenten erlaubt. Andere Speichereinheiten - besonders magnetische Trommeln und Platten - enthalten normalerweise eine Zahl von Spuren, von denen jede eine eigentlich sequentielle Einheit darstellt, die aber gewöhnlich zu kurz ist, um ein ganzes File aufzunehmen. Als Folge davon sind Files auf der Platte gewöhnlich auf mehrere Spuren verteilt und umfassen geeignete Buchhaltungs- information zur Verknüpfung der Spuren. Es ist klar, dass der Anfangspunkt einer jeden Spur eine natürliche Segmentmarke bildet, die leichter direkt angesprochen werden kann als Marken auf einem rein sequentiellen Medium. Eine indizierte Tabelle im Hauptspeicher kann z.B. angeben, welche Spuren mit dem Anfang welcher Segmente zusammenfallen (vgl. Fig. 1.10).

Dies führt uns zu den sogenannten indizierten Files (die manchmal

auch Files mit direktem Zugriff heissen).

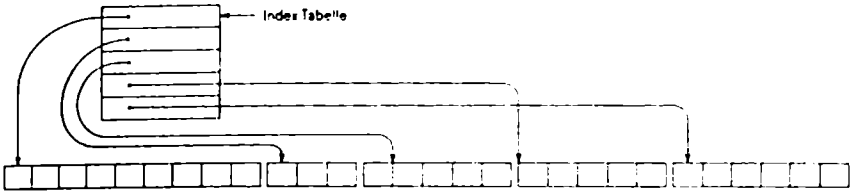


Fig. 1.10: Indiziertes File mit fünf Segmenten

Ueblicherweise sind Trommeln und Platten so organisiert, dass jede Spur mehrere physikalische Marken enthält, an denen Lesen oder Schreiben beginnen kann. Deshalb ist es nicht nötig, dass jedes Segment eine ganze Spur belegt, was eine schwache Speicherausnützung ergeben würde, wenn Segmente kurz sind im Verhältnis zur Spurlänge. Die Speichereinheit zwischen zwei Marken heisst physikalisches Segment (oder Sektor) im Gegensatz zum logischen Segment, das eine sinnvolle Einheit in der Datenstruktur des Programms ist. Sicherlich enthält jedes physikalische Segment höchstens ein logisches Segment, und jedes logische Segment (auch ein leeres) belegt mindestens ein physikalisches Segment. Man sollte daran denken, dass trotz des Namens File mit direktem Zugriff die mittlere Zeit zur Positionierung eines Segments, die sogenannte Latenzzeit, die Hälfte der Umdrehungszeit der Platte beträgt.

Indizierte Files behalten die Charakteristik, dass Schreiben nur an ihrem Ende möglich ist. Sie sind deshalb besonders nützlich, wenn Aenderungen relativ selten erfolgen, weil ~~sie~~ diese meistens das Kopieren des ganzen Files erfordern. Zugreifen kann wesentlich gezielter und schneller über die Indexmarken erfolgen. Dies ist die typische Situation bei sogenannten Datenbanken.

Die Verwendung von Systemen, die selektives Ueberschreiben von Teilen in der Mitte des Files erlauben, ist im allgemeinen schwierig und gefährlich, da die neuen Teile der Information genau gleich lang sein müssen wie die zu ersetzenden alten. Ausserdem ist bei Anwendungen mit grossen Datenmengen selektives Verändern nicht empfehlenswert, da grundsätzlich bei einem Fehler - sei es durch ein falsches Programm, sei es durch ein Versagen der Maschine - ein

Datenzustand zur Verfügung stehen sollte, auf den man zurückgreifen kann, um den durch den Fehler unterbrochenen Prozess zu wiederholen. Daher sollten Aenderungen als Ganzes ausgeführt werden, und zwar so, dass das alte File durch eine auf den neuen Stand gebrachte Kopie ersetzt wird, nachdem eine Prüfung die Richtigkeit des neuen Files ergeben hat. Zum Abändern ist die sequentielle Organisation vom Standpunkt der Zuverlässigkeit aus bei weitem die beste. Sie ist komplizierteren Organisationen grosser Datenmengen vorzuziehen, die zwar effizienter sein können, oft aber zu einem vollständigen Verlust bei einem Defekt der Anlage führen.

1.11.3. Texte

Files mit Komponenten vom Typ char spielen eine besonders wichtige Rolle in der Datenverarbeitung: sie dienen als Verbindung zwischen der Rechenanlage und ihren menschlichen Benützern. Die vom Programmierer abgegebene lesbare Eingabe wie auch die die berechneten Resultate darstellende lesbare Ausgabe sind Folgen von Zeichen. Diesem Datentyp soll daher ein Standardname gegeben werden.

```
type text = file of char .
```

Die Kommunikation zwischen einem Rechenprozess und seinem menschlichen Schöpfer wird letztlich durch ein Bindeglied ermöglicht, das durch zwei Text-Files dargestellt werden kann. Eines enthält die Eingabe (input) für den Rechenprozess, das andere die berechneten Resultate, die sogenannte Ausgabe (output). Wir werden im weiteren die Existenz dieser beiden Files in allen Programmen entsprechend folgender Vereinbarung annehmen:

```
var input, output: text
```

Unter der Annahme, dass diese Files die Standard- Ein- und Ausgabe-Einheiten einer Rechenanlage darstellen (etwa Kartenleser und Zeilendrucker), wollen wir auf dem File input nur das Lesen und auf dem File output nur das Schreiben zulassen.

Da die beiden Standard-Files überaus häufig gebraucht werden, sollen input und output implizit als erste Parameter der Prozeduren

read und write angenommen werden, wenn deren erster Parameter nicht selbst eine File-Variable ist. Ausserdem nehmen wir uns die Freiheit, für beide Standard-Prozeduren eine beliebige Zahl von Argumenten zuzulassen. Diese Vereinbarungen über die Notation lassen sich folgendermassen zusammenfassen:

```
read(x1, ... , xn)   steht für   read(input, x1, ... , xn)
write(x1, ... , xn)  steht für   write(output, x1, ... , xn)

read(f, x1, ... , xn)  steht für
    begin read(f,x1); ... ; read(f,xn) end

write(f, x1, ... , xn)  steht für
    begin write(f,x1); ... ; write(f,xn) end
```

Texte sind typische Beispiele von Sequenzen, die eine Unterstruktur aufweisen. Die üblichen Einheiten der Unterstruktur sind Kapitel, Abschnitte und Zeilen. Zur Darstellung dieser Unterstruktur in Text-Files werden häufig Trennzeichen verwendet. Das Leerzeichen ist das bekannteste Beispiel, aber ähnliche Trennzeichen können verwendet werden, um das Ende von Zeilen, Abschnitten und Kapiteln anzugeben. So enthält z.B. der weitverbreitete ISO Zeichensatz (inklusive seiner amerikanischen Variante ASCII) mehrere solcher sogenannten Steuerzeichen (siehe Anhang).

In diesem Buch wollen wir absehen vom Gebrauch spezieller Trennzeichen und der damit verbundenen Festlegung einer Darstellung der Unterstruktur. Dagegen stellen wir uns einen Text als File vor, das aus Folgen von Zeichen besteht, wobei diese Folgen einzelnen Zeilen entsprechen. Ebenso wollen wir die Diskussion auf eine Stufe einer Unterstruktur beschränken, nämlich auf die der Zeile. Wir betrachten jedoch Texte nicht als File von Files von druckbaren Zeichen, sondern als File von Zeichen und führen zusätzliche Operatoren und Prädikate ein zur Behandlung, d.h. zum Markieren und Erkennen der Zeilenstruktur. Ihre Wirkung versteht man am besten, wenn man annimmt, dass Zeilen durch (hypothetische) Trennzeichen getrennt sind (diese Zeichen gehören jedoch nicht zum Typ char) und dass ihre Aufgabe das Einfügen und Erkennen solcher Trennzeichen ist.

writeln(f) hängt an das File f eine Zeilen-Marke an.

readln(f) überspringt die Zeichen auf dem File f bis zu den direkt auf die nächste Zeilen-Marke folgenden.

eoln(f) ist eine Boolesche Funktion; sie liefert den Wert true, wenn das File auf ein Trennzeichen positioniert ist, sonst hat sie den Wert false. (Für eoln(f) = true soll gelten f↑ = Leerzeichen.)

Wir sind nun in der Lage, zwei Programmschemata für das Schreiben und Lesen von Texten entsprechend dem "Schreiben" und "Lesen" anderer Files (vgl. (1.52) und (1.53)) zu formulieren. Diese Schemata verwenden ein Text-File f und sind ausgerichtet auf das Erstellen und Erkennen der Zeilenstruktur. Sei A(x) eine Anweisung, die x einen Wert (vom Typ char) zuweist, und seien die Bedingungen p und q definiert mit der Bedeutung "dies war das letzte Zeichen der Zeile" und "dies war das letzte Zeichen des Files". Auszuführen seien die Anweisungen U zu Beginn jeder eingelesenen Zeile, S(x) für jedes Zeichen x des Files und V am Ende jeder Zeile.

Schreiben eines Textes f

```
rewrite(f);                                          (1.54)
while not q do
   begin
      while not p do
         begin A(x); write(f,x)
         end;
      writeln(f)
   end
```

Lesen eines Textes f

```
reset(f);                                            (1.55)
while not eof(f) do
   begin U;
      while not eoln(f) do
         begin read(f,x); S(x)
         end;
      V; readln(f)
   end
```

Es gibt Fälle, in denen die Zeilenstruktur eines Textes keine relevante Information darstellt, sondern rein zufällig ist. Unsere Annahme über den Wert der Puffervariablen (entsprechend der Definition von eoln(f)) erlaubt bei einem Trennzeichen in diesem Fall die Verwendung eines einfachen Programmschemas. Jedes Zeilenende erscheint somit als zusätzliches Leerzeichen.

```
while not eof(f) do                                    (1.56)
   begin read(f,x); S(x)
   end
```

In den meisten Programmiersprachen ist es üblich, bei den Prozeduren read und write auch Argumente vom Typ integer oder real zuzulassen. Diese Verallgemeinerung wäre sehr einfach, wenn die Typen integer und real als Array von Zeichen definiert wären, dessen Elemente die einzelnen Ziffern der Zahl darstellen. Stark auf kommerzielle Anwendungen ausgerichtete Sprachen verwenden tatsächlich solche Definitionen und benötigen eine Darstellung von Zahlen in Form von dezimalen Ziffern und des dezimalen Zahlensystems. Das Einführen der Datentypen integer und real als fundamentale Typen hat den wesentlichen Vorteil, dass solche detaillierten Angaben ausgelassen werden können und dass das System verschiedene, für seine Zwecke geeignetere Darstellungen der Zahlen verwenden kann. Tatsächlich benutzen auf wissenschaftliche Berechnungen ausgerichtete Systeme ausnahmslos eine binäre Darstellung, die in fast allen Aspekten der dezimalen Darstellung überlegen ist.

Dies bedeutet aber, dass der Programmierer nicht annehmen kann, Zahlen direkt von Text-Files ohne zugehörige Konversionsoperationen lesen oder schreiben zu können. Es ist üblich, diese Konversionsoperationen hinter read- und write-Anweisungen mit numerischen Argumenten zu verbergen. Der professionnelle Programmierer ist sich aber bewusst, dass solche Anweisungen (sogenannte E/A-Anweisungen) zwei verschiedene Funktionen ausüben: den Transport von Daten zwischen verschiedenen Speichermedien und die Umwandlung der Datendarstellung. Das letztere kann sehr wohl kompliziert und zeitaufwendig sein.

In den folgenden Kapiteln dieses Buches werden die read- und write-Anweisungen mit numerischen Argumenten nach den Regeln der

Programmiersprache PASCAL verwendet. Diese Regeln erlauben eine gewisse Art der Formatspezifikation zur Steuerung des Umwandlungsprozesses. Die Formatspezifikation gibt die Zahl der gewünschten Ziffern im Fall von write-Anweisungen an. Diese Anzahl von Zeichen - auch "Feldlänge" genannt - wird unmittelbar an das Argument angehängt:

```
write(f,x:n)
```

Das Argument x ist auf das File f zu schreiben; sein Wert ist in eine Folge von (mindestens) n Zeichen umzuwandeln; wenn notwendig, wird den Ziffern ein Vorzeichen und eine entsprechende Zahl von Leerzeichen vorangestellt.

Weitere Details sind für das Verständnis der späteren Programmbeispiele dieses Buches nicht notwendig. Zwei Beispiele von Routinen zur Konversion der Zahlendarstellung sind hier angeführt (Programme 1.3 und 1.4), um die Komplexität solcher Operationen zu zeigen, die gewöhnlich implizit in der write-Anweisung ausgeführt werden. Die zwei Prozeduren des PASCAL 6000 Systems zeigen die Umwandlung reeller Zahlen von ihrer dezimalen Darstellung in eine beliebige "interne" Darstellung und umgekehrt. (Die Konstanten im Prozedur-Kopf sind bestimmt durch die Eigenschaften des Formates der Gleitkommazahlen der Rechenanlage CDC 6000: 11 Bit binärer Exponent und 48 Bit Mantisse.)

```
procedure readreal(var f: text; var x: real);
   {Einlesen einer reellen Zahl x vom File f}
   {folgende Konstanten sind systemabhängig}
   const t48 = 281474976710656;    {48 Bit Mantisse}
      limit = 56294995342131;    {= t48 div 5}
      z = 27;      {= ord('0') }
      lim1 = 322;      { grösster Exponent }
      lim2 = -292;     { kleinster Exponent }
   type posint = 0..323;
   var ch: char; y: real; e,i,e: integer;
      s,ss: boolean;    { Vorzeichen }

   function ten(e: posint): real;  { = 10**e, 0<e<322 }
      var i: integer; t: real;
```

```
   begin i := 0; t := 1.0;
      repeat if odd(e) then
         case i of
          0: t := t * 1.0e1;
          1: t := t * 1.0e2;
          2: t := t * 1.0e4;
          3: t := t * 1.0e8;
          4: t := t * 1.0e16;
          5: t := t * 1.0e32;
          6: t := t * 1.0e64;
          7: t := t * 1.0e128;
          8: t := t * 1.0e256
         end :
         e := e div 2; i := i+1
      until e = 0;
      ten := t
   end :

begin
   if eof(f) then
   begin  message(' tried to read past end of file f');
          halt
   end;
   {überspringe führende Leerzeichen}
   while (f↑=' ') end (not eof(f)) do get(f);
   if not eof(f) then
   begin
      ch := f↑;
      if ch = '-' then
         begin s := true; get(f); ch := f↑
         end else
         begin s := false;
            if ch = '+' then
            begin get(f); ch := f↑
            end
         end :
      if not (ch in ['0'..'9']) then
      begin message(' digit expected'); halt;
      end;
      e := 0; e := 0;
```

```
repeat if e < limit then e := 10*e + ord(ch)-z
                    else e := e+1;
   get(f); ch := f↑
until not (ch in ['0'..'9']);
if ch = '.' then
begin { Einlesen des Bruches } get(f); ch := f↑;
   while ch in ['0'..'9'] do
   begin if e < limit then
         begin e := 10*e + ord(ch)-z; e := e-1
         end :
      get(f); ch := f↑
   end
end :
if ch = 'e' then
begin { Einlesen des Skalierungsfaktors }
   i := 0; get(f); ch := f↑;
   if ch = '-' then
   begin ss := true; get(f); ch := f↑
   end else
   begin ss := false; if ch = '+' then
      begin get(f); ch := f↑
      end
   end :
   if ch in ['0'..'9'] then
   begin i := ord(ch)-z; get(f); ch := f↑;
      while ch in ['0'..'9'] do
      begin if i < limit then i := 10*i + ord(ch)-z;
         get(f); ch := f↑
      end
   end else
   begin message(' digit expected'); halt
   end :
   if ss then e := e-i else e := e+i;
end :
if e < lim2 then
   begin e := 0; e := 0
   end else
if e > lim1 then
begin message(' number too large'); halt end;
{ 0 < e < 2**49 }
```

```
      if e >= t48 then y := ((e+1) div 2) * 2.0 else y := e;
      if s then y := -y;
      if e < 0 then x := y/ten(-e) else
      if e <> 0 then x := y*ten(e) else x := y;
   end;
end {readreal}
```

Programm 1.3

```
procedure writereal(var f: text; x: real; n: integer);
   {Schreiben einer reellen Zahl x mit n Zeichen in dezimalem
    Gleitkommaformat}
   {Die folgenden Konstanten hängen von der zugrundeliegenden
    Darstellung der Gleitkommazahlen ab}
   const t48 = 281474976710656;
            { = 2**48; 48 = Grösse der Mantisse }
         z = 27;        { ord('0') }
   type posint = 0..323;    { Bereich des dezimalen Exponents }
   var c,d,e,e0,e1,e2,i: integer;

   function ten(e: posint): real;    { 10**e, 0<e<322 }
      var i: integer; t: real;
   begin i := 0; t := 1.0;
      repeat if odd(e) then
         case i of
          0: t := t * 1.0e1;
          1: t := t * 1.0e2;
          2: t := t * 1.0e4;
          3: t := t * 1.0e8;
          4: t := t * 1.0e16;
          5: t := t * 1.0e32;
          6: t := t * 1.0e64;
          7: t := t * 1.0e128;
          8: t := t * 1.0e256
         end :
         e := e div 2; i := i+1
      until e = 0;
      ten := t
   end { ten } :
```

```
begin { es werden zumindest 10 Zeichen benötigt: b+9.9e+999 }
   if x = 0 then
   begin  repeat write(f, ' '); n := n-1
          until n <= 1;
      write(f, '0')
   end else
   begin
      if n <= 10 then n := 3 else n := n-7;
      repeat write(f, ' '); n := n-1
      until n <= 15;
      { 1 < n <= 15,  Zahl der zu druckenden Ziffern }
      begin { teste Vorzeichen und ermittle Exponent }
         if x < 0 then
            begin write(f, '-'); x := -x
            end else write(f, ' ');
         e := expo(x);  {e = entier(log2(abs(x)))}
         if e >= 0 then
            begin e := e*77 div 256 +1; x := x/ten(e);
               if x >= 1.0 then
                  begin x := x/10.0; e := e+1
                  end
            end else
            begin e := (e+1)*77 div 256; x := ten(-e)*x;
               if x < 0.1 then
                  begin x := 10.0*x; e := e-1
                  end
            end ;
         { 0.1 <= x < 1.0 }
         case n of       { Runden }
           2: x := x+0.5e-2;
           3: x := x+0.5e-3;
           4: x := x+0.5e-4;
           5: x := x+0.5e-5;
           6: x := x+0.5e-6;
           7: x := x+0.5e-7;
           8: x := x+0.5e-8;
           9: x := x+0.5e-9;
          10: x := x+0.5e-10;
          11: x := x+0.5e-11;
```

```
            12: x := x+0.5e-12;
            13: x := x+0.5e-13;
            14: x := x+0.5e-14;
            15: x := x+0.5e-15
          end;
          if x >= 1.0 then
             begin x := x * 0.1; e := e+1;
             end ;
          c := trunc(x,48);
          c := 10*c; d := c div t48;
          write(f, chr(d+z), '.');
          for i := 2 to n do
          begin c := (c - d*t48) * 10; d := c div t48;
             write(f, chr(d+z))
          end ;
          write(f, 'e'); e := e-1;
          if e < 0 then
             begin write(f, '-'); e := -e
             end  else write(f, '+');
          e1 := e * 205 div 2048; e2 := e - 10*e1;
          e0 := e1 * 205 div 2048; e1 := e1 - 10*e0;
          write(f, chr(e0+z), chr(e1+z), chr(e2+z))
       end
    end
end {writereal}
```

Programm 1.4

1.11.4. Ein Programm zum Aendern eines Files

Als Beispiel für eine Anwendung sequentieller Strukturen stellen wir ein spezielles Problem, das ausserdem dazu dienen soll, eine Methode zur Entwicklung und Erläuterung von Programmen zu zeigen. Diese Methode heisst schrittweise Verfeinerung [32] und wird zur Erklärung der meisten Algorithmen in diesem Buch verwendet.

Das Problem ist folgendes: es soll ein Programm entwickelt werden, das einen Text x in einen Text y umformt. Unter Aendern (Editing) verstehen wir das Streichen oder Ersetzen bestimmter Zeilen oder

das Einfügen neuer Zeilen. Die Aenderungen werden durch eine Reihe von Instruktionen im Standard-Text-File input angegeben:

I,m.	Einschub eines Textes nach der m-ten Zeile.
D,m,n.	Streichen der Zeilen m bis n.
R,m,n.	Ersetzen der Zeilen m bis n.
E.	Beenden des Aenderungsprozesses.

Jede Instruktion stellt eine Zeile im Standard-File input dar, das wir Instruktions-File nennen. m und n sind dezimale Zeilennummern. Einzuschiebende Texte folgen unmittelbar auf die Befehle I und R und werden durch eine leere Zeile abgeschlossen.

Die Instruktionen zur Aenderung des Files sollen mit zunehmenden Zeilennummern eingegeben werden. Diese Regel erlaubt ein rein sequentielles Verarbeiten des Eingabe-Textes x. Es ist klar, dass der Zustand des Prozesses durch die gegenwärtige Position von x, d.h. durch die Nummer der gerade untersuchten Zeile zu charakterisieren ist.

Nehmen wir an, dass das Aenderungsprogramm interaktiv verwendet werden soll und dass deshalb das Instruktions-File durch die von der Konsole eingegebenen Daten dargestellt wird. Bei einer solchen Verarbeitung möchte man irgendeine Art von Rückmeldungen haben. Eine geeignete und nützliche Rückmeldung ist der Text der Zeile, auf die durch die letzte Instruktion positioniert wurde. Diese Zeile nennen wir gegenwärtige Zeile. Die neue Forderung, das File nach jeder Instruktion zu drucken, bedingt die Darstellung der gegenwärtigen Zeile durch eine explizite Variable, in der die Zeile gespeichert ist, nachdem sie von x gelesen wurde und bevor sie auf y geschrieben wird. Diese Technik heisst "Pufferung". Das Aenderungsprogramm kann folgendermassen formuliert werden:

```
var lno: integer; {Nummer der gegenwärtigen Zeile}        (1.57)
    cl : line;    {gegenwärtige Zeile}
    x,y: text;

begin "lese eine Instruktion";
      repeat "interpretiere die Instruktion";
             "schreibe die Zeile";
```

```
            "lese eine Instruktion"
        until "Instruktion" = E
    end
```

Wir wollen nun weitere Details der verschiedenen Anweisungen schrittweise ausarbeiten. Beim Verfeinern von "lese eine Instruktion" und "interpretiere die Instruktion" erkennen wir, dass ein Befehl im allgemeinen drei Teile umfasst: den Befehls-Code und zwei Parameter. Zur Verbindung zwischen diesen beiden Routinen führen wir daher die drei Variablen code, m und n ein.

```
var code,ch: char;
    m,n: integer
```

"lese eine Instruktion":

```
read(code,ch);                                              (1.58)
if ch = ',' then read(m,ch) else m := lno;
if ch = ',' then read(n) else n := m
```

Diese Formulierung kann 0, 1 oder 2 Parameter akzeptieren, indem geeignete Annahmen für die "fehlenden" Spezifikationen getroffen werden.

"interpretiere die Instruktion"

```
copy;                                                       (1.59)
if code = 'I' then
begin putline; insert
end else
if code = 'D' then skip else
if code = 'R' then
begin insert; skip
end else
if code = 'E' then copyrest else "Fehler"
```

In einem zweiten Schritt der Verfeinerung drücken wir die in (1.59) vorkommenden Anweisungen copy, insert und skip durch Operationen aus, die nur einzelne Zeilen umfassen, d.h. durch getline und putline. Ihre gemeinsame Charakteristik ist die Struktur der

Wiederholung. Copy kopiert Zeilen von x auf y, wobei mit der gegenwärtigen Zeile begonnen und mit der m-ten Zeile aufgehört wird. Skip liest Zeilen von x bis zur n-ten Zeile, ohne sie zu kopieren.

```
copy:      while lno < m do                                   (1.60)
           begin putline; getline
           end

skip:      while lno < n do getline

insert:    readline;
           while noend do
              begin putline; readline
              end;
           getline

copyrest:  while not eof(x) do
              begin putline; getline
              end;
           putline
```

Im dritten und letzten Schritt der Verfeinerung drücken wir die Operationen getline, putline, readline und writeline aus durch Operationen auf einzelnen Zeichen. Wir merken, dass bis jetzt alle Operationen nur mit ganzen Zeilen gearbeitet haben und dass keine speziellen Annahmen über die genaue Unterstruktur einer Zeile gemacht wurden. Wir wissen, dass Zeilen selbst Folgen von Zeichen sind. Man könnte versucht sein, die Variable c1 (die die gegenwärtige Zeile enthält) als eine Sequenz zu definieren

```
var c1: file of char .
```

Aber erinnern wir uns an die Empfehlung, dass eine Struktur mit unendlicher Kardinalität nie verwendet werden soll, wenn eine einfache Struktur (wie ein Array) genügt. Wir sind tatsächlich wohlberaten, im vorliegenden Fall eine Array-Struktur zu verwenden. Dies ist leicht durchführbar, wenn durch die Problemspezifikation die Zeilenlänge auf einen festen Wert beschränkt ist. Wir spezifizieren dann zum Beispiel

```
var c1: array [1..80] of char .
```

Die vier Routinen benutzen im Zusammenhang mit diesem Array eine Variable i, die in Wirklichkeit lokal verwendet wird und daher ebenso gut lokal zu jeder Routine vereinbart werden kann; ausserdem wird es nötig, eine globale Variable L einzuführen, die die Länge der gegenwärtigen Zeile angibt.

```
getline:    i := 0; lno := lno + 1;                          (1.61)
            while not eoln(x) do
               begin i := i+1; a[i] := x↑; get(x)
               end;
            L := i; getln(x)

putline:    i := 0;
            while i < L do
               begin i := i+1; y↑ := a[i]; put(y)
               end;
            putln(y)

readline:   i:= 0;
            while not eoln(input) do
               begin i := i+1; read(a[i])
               end;
            readln

writeline:  i := 0;
            while i < L do
               begin i := i+1; write(a[i])
               end;
            writeln
```

Die Bedingung noend in der Routine insert kann nun ausgedrückt werden durch

L <> 0 .

Damit ist die Entwicklung des Programms zur Aenderung eines Files abgeschlossen.

Uebungen

1.1. Angenommen die Kardinalitäten der Standard-Typen integer, real und char seien mit kI, kR und kC bezeichnet. Wie gross sind dann die Kardinalitäten der folgenden Datentypen, die als Beispiele in diesem Kapitel definiert werden:

Geschlecht, Boolean, Werktag, Buchstabe, Ziffer, Offizier, Zeile, alfa, Komplex, Datum, Person, Koordinate, Zeichensatz, Bandzustand.

1.2. Wie würden Sie Variable dieser Typen darstellen

(a) Im Speicher Ihres Computers?
(b) In FORTRAN?
(c) In der Ihnen geläufigen Programmiersprache?

1.3. Wie lauten die Befehlsfolgen (auf Ihrem Computer) für
(a) Zugreifen und speichern von Elementen von gepackten Records und Arrays?
(b) Mengenoperationen inklusive Operator in?

1.4. Kann die richtige Verwendung von varianten Records zur Ausführungszeit geprüft werden? Kann sie sogar zur Compilationszeit geprüft werden?

1.5. Welche Gründe führen zur Definition von gewissen Datenmengen als sequentielle Files anstelle von Arrays?

1.6. Nehmen wir an, Sie sollten sequentielle Files entsprechend der Definition in Abschnitt 1.11 auf einem Computer mit sehr grossem Hauptspeicher implementieren. Es sei Ihnen erlaubt, Files auf eine gewisse Länge L zu beschränken, d.h. Sie können Files in Form von Arrays darstellen.

Beschreiben Sie eine mögliche Implementation inklusive der gewählten Datendarstellung und Prozeduren für die elementaren File-Operatoren get, put, reset und rewrite, die durch die Axiome in Abschnitt 1.11 definiert sind.

1.7. Wenden Sie Aufgabe 1.6 auf den Fall segmentierter Files an.

1.8. Gegeben sei ein Fahrplan, der die täglichen Verbindungen auf verschiedenen Linien eines Eisenbahnnetzes angibt. Zu finden ist eine Darstellung dieser Daten in Form von Arrays, Records oder Files, die sich für das Nachschlagen von Ankunfts- und Abfahrtszeiten an einer gewissen Station und für eine gewünschte Zugverbindung eignet.

1.9. Gegeben sei ein Text T in Form eines File sowie Listen von einigen wenigen Worten in Form von zwei Arrays A und B. Worte seien kurze Arrays von Zeichen mit kleiner konstanter Maximallänge.

Zu schreiben ist ein Programm, das den Text T in einen Text S umformt, wobei jedes in T auftretende Wort A[i] durch das entsprechende Wort B[i] ersetzt wird.

1.10. Welche Aenderungen - Neudefinition von Konstanten usw. - sind nötig, um Programme 1.3 und 1.4 an den Ihnen zur Verfügung stehenden Computer anzupassen?

1.11. Schreiben Sie eine Prozedur analog zu Programm 1.4 mit dem Prozedur-Kopf

```
procedure writereal(var f: text; x: real; n,m: integer);
```

Sie soll den Wert x umformen in eine Folge von mindestens n Zeichen (die an das File f anzuhängen sind), die x in dezimaler Festkommaform mit m Ziffern nach dem Dezimalpunkt darstellen. Wenn nötig, soll der Zahl eine geeignete Anzahl Leerzeichen und/oder ein

Vorzeichen vorausgehen.

1.12. Das Programm zur Textänderung von Abschnitt 1.11.4 ist als vollständiges Programm zusammenzufassen.

1.13. Zu vergleichen sind die folgenden drei Versionen des binären Suchens nach (1.17). Welche der drei Programme sind richtig? Welche sind effizienter?

Gegeben seien die folgenden Variablen und eine Konstante N > 0 :

```
var i,j,k: integer;
    a: array [1..N] of T;
    x: T
```

Programm A:

```
i := 1; j := N;
repeat k := (i+j) div 2;
   if a[k] < x then i := k else j := k
until (a[k]=x) or (i >= j)
```

Programm B:

```
i := 1; j := N;
repeat k := (i+j) div 2;
   if x <= a[k] then j := k-1;
   if a[k] <= x then i := k+1
until i > j
```

Programm C:

```
i := 1; j := N;
repeat k := (i+j) div 2;
   if x < a[k] then j := k else i := k+1
until i >= j
```

Hinweis: Alle Programme müssen abschliessen mit a[k] = x, wenn ein solches Element existiert, oder a[k] <> x, wenn kein Element mit dem Wert x existiert.

1.14. Eine Gesellschaft führt eine Umfrage zur Bestimmung des Erfolgs ihrer Produkte durch. Die Produkte sind Schlagerplatten und -bänder, und die beliebtesten Schlager sollen in einer Hit- parade gesendet werden. Die befragte Bevölkerung ist entsprechend Geschlecht und Alter (bis und mit 20 oder älter) in vier Kategorien einzuteilen. Jede Person soll fünf Schlager wählen. Schlager werden durch Zahlen von 1 bis N bezeichnet (z.B. N = 30). Das Ergebnis der Umfrage liegt in Form des Files "poll" vor.

```
type hit = 1 .. N;
     sex = (male, female);
     response =
        record name, firstname: alfa;
               s: sex;
               age: integer;
               choice: array[1..5] of hit
        end ;
var  poll: file of response
```

Somit entspricht jedes File-Element einem Befragten und enthält seinen Namen, Vornamen, Geschlecht, Alter und die fünf von ihm bevorzugten Schlager in der Reihenfolge der Beliebtheit. Dieses File ist die Eingabe zu einem Programm, das folgende Resultate liefern soll:

1. Eine Liste von Schlagern in der Reihenfolge ihrer Beliebtheit. Jeder Eintrag besteht aus der Nummer des Schlagers und der Anzahl der in der Umfrage erhaltenen Stimmen. Nie genannte Schlager werden in der Liste nicht aufgeführt.

2. Vier getrennte Listen mit den Namen und Vornamen aller Befragten, die an erster Stelle einen der drei in ihrer Kategorie beliebtesten Schlager genannt haben.

Die fünf Listen sind mit geeigneten Ueberschriften zu versehen.

2. Sortieren

2.1. Einleitung

Dieses Kapitel enthält in der Hauptsache eine ausgiebige Menge von Beispielen, die die Verwendung der im vorangehenden Kapitel behandelten Datenstrukturen erläutern und zeigen, wie stark die Wahl der Struktur der zugrunde liegenden Daten die Algorithmen beeinflusst, die eine bestimmte Aufgabe ausführen. Sortieren ist auch ein gutes Beispiel, um zu zeigen, dass eine Aufgabe mit vielen verschiedenen Algorithmen gelöst werden kann, wovon jeder gewisse Vor- und Nachteile hat, die alle je nach der besonderen Anwendung gegeneinander abzuwägen sind.

Unter Sortieren versteht man allgemein den Prozess des Anordnens einer gegebenen Menge von Objekten in einer bestimmten Ordnung. Der Sinn des Sortierens liegt in der Vereinfachung der späteren Suche nach Elementen in der geordneten Menge. In diesem Zusammenhang ist das Sortieren eine fast universell angewandte, grundlegende Tätigkeit. Objekte sind sortiert in Telefonbüchern, Steuerlisten, Inhaltsangaben, Büchereien, Wörterbüchern, Lagerhäusern und fast überall, wo gespeicherte Objekte gesucht und wiedergefunden werden müssen. Sogar kleinen Kindern wird gelehrt, ihre Sachen zu ordnen, und sie werden mit einer gewissen Art von Sortieren bereits konfrontiert, lange bevor sie etwas über Arithmetik lernen.

Demnach ist Sortieren besonders in der Datenverarbeitung eine wichtige und wesentliche Tätigkeit. Was wäre einfacher zu sortieren als "Daten"! Trotzdem richten wir beim Sortieren unser hauptsächliches Augenmerk auf noch grundlegendere Techniken, die bei der Konstruktion von Algorithmen verwendet werden. Es gibt nicht viele Programmierprobleme, die nicht in irgendeiner Weise in Verbindung mit den Sortier-Algorithmen vorkommen. Ganz besonders ist Sortieren ein ideales Beispiel, um eine grosse Vielfalt von Algorithmen aufzuzeigen, alle mit dem gleichen Zweck, viele in gewisser Hinsicht optimal und die meisten mit Vorteilen gegenüber anderen. Es ist deshalb auch ein ideales Thema, die Notwendigkeit der Leistungs-

analyse von Algorithmen darzulegen. Am Beispiel des Sortierens kann man ausserdem ganz gut den sehr bedeutenden Leistungszuwachs zeigen, der durch die Entwicklung raffinierter Algorithmen erzielt werden kann.

Die Wahl eines Algorithmus hängt beim Sortieren so stark von der Struktur der zu bearbeitenden Daten ab - ein häufig auftretendes Phänomen - dass Sortiermethoden allgemein in zwei Kategorien eingeteilt werden, nämlich Sortieren von Arrays und Sortieren von (sequentiellen) Files. Beide Arten werden oft internes und externes Sortieren genannt, da die Arrays in dem schnellen, willkürlich zugreifbaren "internen" Speicher der Rechenanlagen gespeichert sind, während Files zweckmässig in den langsameren, jedoch grösseren "externen" Speichern untergebracht sind, die auf mechanisch bewegten Vorrichtungen beruhen (Platten und Bänder). Die Bedeutung dieser Unterscheidung geht aus dem Beispiel des Sortierens numerierter Karten hervor (Fig. 2.1). Das Strukturieren der Karten als ein Array entspricht ihrem Auslegen vor dem Sortierenden, so dass jede Karte sichtbar und einzeln greifbar ist.

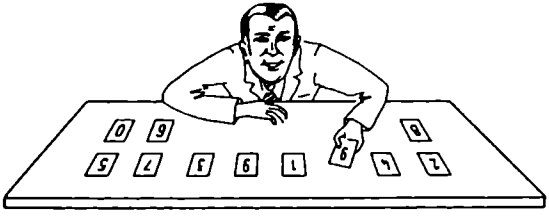


Fig. 2.1: Sortieren eines Array

Das Strukurieren der Karten als File (Fig. 2.2) bedeutet jedoch, dass von jedem Stapel nur die oberste Karte sichtbar ist. Solch eine Einschränkung wird natürlich für die zu verwendende Sortiermethode wesentliche Konsequenzen haben, ist aber nicht zu vermeiden, wenn die Zahl der auszulegenden Karten grösser ist als der zur Verfügung stehende Tisch.

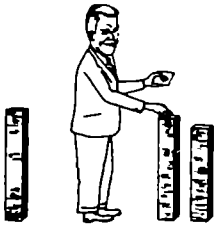


Fig. 2.2: Sortieren eines Files

Bevor wir weiterfahren, führen wir eine im ganzen Kapitel verwendete Terminologie und Notation ein. Gegeben seien die Elemente

$$a[1],\ a[2],\ \ldots\ ,\ a[n]\ .$$

Das Sortieren besteht im Vertauschen dieser Elemente in eine Reihenfolge

$$a[k1],\ a[k2],\ \ldots\ ,\ a[kn]\ ,$$

so dass für eine gegebene Ordnungsfunktion f gilt

$$f(a[k1]) <= f(a[k2]) <= \ldots <= f(a[kn]) \qquad (2.1)$$

Gewöhnlich wird die Ordnungsfunktion nicht nach einer bestimmten Berechnungsvorschrift berechnet, sondern als explizite Komponente (Feld) eines jeden Elementes (item) gespeichert. Sein Wert heisst Schlüssel (key) des Elementes. Folglich ist die Strukturart Record für die Darstellung der Elemente a[i] besonders gut geeignet. Wir definieren deshalb einen Typ item, wie er in allen folgenden Sortier-Algorithmen verwendet wird:

```
type item = record key: integer;                    (2.2)
                [andere Komponenten sind hier aufzuführen]
            end
```

Die "anderen Komponenten" stellen wichtige Daten bezüglich der Elemente der Sammlung dar; der Schlüssel dient nur zur Indentifizierung der Elemente. Für unsere Sortier-Algorithmen ist jedoch der

Schlüssel die einzig wichtige Komponente, und es ist nicht notwendig, irgendwelche weiteren Komponenten zu definieren. Die Wahl von integer als Schlüsseltyp ist etwas willkürlich. Natürlich könnte ebenso gut jeder andere Typ verwendet werden, für den eine vollständige Ordnungsrelation definiert ist.

Eine Sortiermethode heisst stabil, wenn die relative Ordnung der Elemente mit gleichen Schlüsseln durch den Sortierprozess unverändert bleibt. Stabilität beim Sortieren ist oft erwünscht, wenn die Elemente bereits nach einem zweitrangigen Schlüssel geordnet (sortiert) sind, d.h. nach Eigenschaften, die nicht durch den (Haupt-) Schlüssel selbst ausgedrückt werden.

Dieses Kapitel soll nicht als umfassende Uebersicht über Sortiertechniken betrachtet werden, es behandelt vielmehr einige besondere Methoden ausführlicher. Für eine eingehende Behandlung des Themas Sortieren wird der interessierte Leser auf das ausgezeichnete und umfassende Kompendium von D.E. Knuth [18] verwiesen (siehe auch [19]).

2.2. Sortieren von Arrays

Die wichtigste Voraussetzung bei Sortiermethoden für Arrays ist eine wirtschafliche Verwendung des vorhandenen Speichers. Dies bedeutet, dass die Umstellung zur Ordnung der Elemente am Ort (in situ) auszuführen ist und dass Methoden, die Elemente von einem Array a zu einem Resultat-Array b transportieren, nicht sehr interessant sind. Wenn wir so unsere Wahl unter den vielen möglichen Methoden durch das Kriterium der Wirtschaftlichkeit des Speichereinsatzes einschränken, kommen wir zu einer ersten Klassifikation nach ihrer Effizienz, d.h. ihrem Zeitaufwand. Ein gutes Mass für Effizienz bilden die Anzahl C der erforderlichen Schlüsselvergleiche und die Anzahl M der Bewegungen (Umstellungen) der Elemente. Diese Zahlen sind Funktionen der Anzahl n der zu sortierenden Elemente.

Während gute Sortier-Algorithmen O(n*log n) Vergleiche benötigen,

erörtern wir zuerst verschiedene einfache und leicht verständliche Techniken des Sortierens, sogenannte direkte Methoden, die alle in der Grössenordnung n**2 Schlüsselvergleiche benötigen. Es gibt drei gute Gründe, die direkten Methoden vor den schnelleren Algorithmen zu behandeln.

1. Die direkten Methoden sind besonders gut geeignet, um die Eigenschaften der wesentlichen Prinzipien des Sortierens darzulegen.

2. Ihre Programme sind kurz und leicht verständlich. Man beachte, dass auch Programme Speicherplatz benötigen!

3. Obwohl höher entwickelte Methoden weniger Operationen erfordern, sind diese Operationen im Detail gewöhnlich komplexer; folglich sind die direkten Methoden für hinreichend kleines n schneller, jedoch sind sie für grosses n nicht zu verwenden.

Sortiermethoden, die Elemente am Ort sortieren, können je nach der zugrunde liegenden Methode in drei Hauptkategorien eingeteilt werden:

Sortieren durch Einfügen (insertion)
Sortieren durch Auswählen (selection)
Sortieren durch Austauschen (exchange).

Diese drei Prinzipien sollen nun untersucht und verglichen werden. Die Programme arbeiten auf der Variablen a, deren Komponenten am Ort zu sortieren sind, und beziehen sich auf die in (2.2) und (2.3) definierten Datentypen item und index.

```
type index = 0..n;                                    (2.3)
var a: array [1..n] of item
```

2.2.1. Sortieren durch direktes Einfügen

Diese Methode wird von Kartenspielern mit Vorliebe benutzt. Die Elemente (Karten) werden begrifflich in eine Ziel-Sequenz a[1] ... a[i-1] und eine Quellen-Sequenz a[i] ... a[n] aufgeteilt. Beginnend mit i = 2, wird bei jedem Schritt das i-te Element der Quellen-Se-

quenz herausgegriffen und in die Ziel-Sequenz an die entsprechende Stelle eingefügt und i um 1 erhöht.

Der Vorgang des Sortierens durch Einfügen wird anhand eines Beispiels in Tabelle 2.1 mit acht willkürlich gewählten Zahlen gezeigt.

ursprüngliche Schlüssel:	44	55	12	42	94	18	06	67
i=2	44	55	12	42	94	18	06	67
i=3	12	44	55	42	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	42	44	55	94	18	06	67
i=6	12	18	42	44	55	94	06	67
i=7	06	12	18	42	44	55	94	67
i=8	06	12	18	42	44	55	67	94

Tabelle 2.1

Der Algorithmus des direkten Einfügens lautet

```
for i := 2 to n do
   begin x := a[i];
      "füge x am entsprechenden Platz in a[1]...a[i] ein"
   end
```

Zum Bestimmen des entsprechenden Platzes ist es angebracht, zwischen Vergleichen und Bewegungen abzuwechseln, d.h. x "nach links wandern" zu lassen, mit dem nächsten Element a[j] zu vergleichen und entweder x einzufügen oder a[j] nach rechts zu schieben und nach links fortzufahren. Wir erkennen zwei verschiedene Bedingungen, die den Prozess des "nach links Wanderns" beenden können:

1. Ein Element a[j] mit grösserem Schlüssel als x wird gefunden.
2. Das linke Ende der Ziel-Sequenz ist erreicht.

Dieser typische Fall einer Wiederholung mit zwei Abbruchbedingungen lenkt unsere Aufmerksamkeit auf die bekannte Technik der Verwendung einer Marke. Sie wird hier durch Setzen eines Markierungselementes a[0] := x mühelos realisiert. (Man beachte, dass dazu der Indexbereich in der Vereinbarung von a auf 0..n zu erweitern ist.) Damit

erhalten wir das vollständige Programm 2.1:

```
procedure straigtinsertion;
   var i,j: index;  x: item;
begin
   for i := 2 to n do
   begin x := a[i]; a[0] := x; j := i-1;
      while x.key < a[j].key do
         begin a[j+1] := a[j]; j := j-1
         end ;
      a[j+1] := x
   end
end
```

Programm 2.1

Analyse des direkten Einfügens: Die Zahl C[i] der Vergleiche von Schlüsseln beim i-ten Durchlauf ist höchstens i-1 und mindestens 1 und somit - unter der Annahme, dass alle Permutationen gleich wahrscheinlich sind - im Mittel i/2. Die Zahl M[i] der Bewegungen (Zuweisungen von Elementen) ist C[i]+2 (inklusive Marke). Deshalb sind die Gesamtzahlen der Vergleiche und Bewegungen

```
C[min] = n-1                  M[min] = 3*(n-1)              (2.4)
C[mit] = (n**2+n-2)/4         M[mit] = (n**2+9*n-10)
C[max] = (n**2+n)/2           M[max] = (n**2+3*n-4)/2
```

Die kleinsten Zahlen kommen dann vor, wenn die Elemente von Anfang an geordnet sind; die schlimmsten Fälle treten ein, wenn die Elemente zu Beginn in umgekehrter Reihenfolge angeordnet sind. In diesem Sinn zeigt Sortieren durch Einfügen ein wirklich natürliches Verhalten. Es ist kler, dass der gegebene Algorithmus auch einen stabilen Sortierprozess beschreibt: die Reihenfolge der Elemente mit gleichen Schlüsseln bleibt unverändert.

Der Algorithmus des direkten Einfügens lässt sich ohne weiteres verbessern, indem die bereits vorhandene Ordnung der Ziel-Sequenz a[1]...a[i-1], in die das neue Element einzufügen ist, ausgenutzt wird. Dazu kann eine schnellere Methode zur Bestimmung der Einschubstelle verwendet werden. Auf der Hand liegt eine binäre

Suche, die in der Mitte der Ziel-Sequenz prüft und so lange halbiert, bis die Einschubstelle gefunden ist. Der modifizierte Sortier-Algorithmus heisst binäres Einfügen und wird in Programm 2.2 gezeigt.

```
procedure binaryinsertion;
   var i, j, l, r, m: index;  x: item;
begin
   for i := 2 to n do
   begin x := a[i]; l := 1; r := i-1;
      while l <= r do
      begin m := (l+r) div 2;
         if x.key < a[m].key then r := m-1 else l := m+1
      end ;
      for j := i-1 downto l do a[j+1] := a[j];
      a[l] := x;
   end
end
```

Programm 2.2

Analyse des binären Einfügens: Die Einschubstelle ist gefunden, wenn a[j].key <= x.key < a[j+1].key . Somit muss das Suchintervall am Ende 1 sein, und dazu ist das Intervall mit i Schlüsseln log2(i) mal zu halbieren, und es gilt

$$C = \sum_{i=1}^{n} \text{ganz}(\log 2\ i)$$

Wir approximieren diese Summe durch das Integral

$$\int_1^n \log x\, dx = x(\log x - c) \Big|_1^n = n(\log n - c) + c \ , \qquad (2.5)$$

wobei c = log e = 1/ln 2 = 1.44269... . Die Zahl der Vergleiche ist im wesentlichen unabhängig von der ursprünglichen Reihenfolge der Elemente. Aber, bedingt durch das Abschneiden bei der in der Bisektion des Suchintervalls vorkommenden ganzzahligen Division,

kann die Zahl der für i Elemente wirklich benötigten Vergleiche um 1 grösser sein als erwartet. Durch diese Asymmetrie werden Einschubstellen am unteren Ende im Mittel etwas schneller gefunden als am oberen Ende. Dadurch sind die Fälle begünstigt, bei denen die Elemente ursprünglich völlig ungeordnet sind. Tatsächlich wird das Minimum an Vergleichen benötigt, wenn die Elemente anfangs in umgekehrter Reihenfolge sind, und das Maximum, wenn sie bereits geordnet sind. Folglich ist dies ein Fall von unnatürlichem Verhalten eines Sortier-Algorithmus.

$$C \approx n*(\log n - \log e +- 0.5)$$

Leider gilt die durch eine binäre Suchmethode erreichte Verbesserung nur für die Zahl der Vergleiche, aber nicht für die Zahl der notwendigen Verschiebungen. Da das Verschieben von Elementen, d.h. von Schlüsseln und zugehöriger Information, im allgemeinen wesentlich zeitaufwendiger ist als der Vergleich von zwei Schlüsseln, ist die Verbesserung keineswegs beträchtlich: der wesentliche Faktor M ist immer noch von der Ordnung n**2. Tatsächlich benötigt das Sortieren eines bereits geordneten Array mehr Zeit als direktes Einfügen mit sequentieller Suche! Dieses Beispiel zeigt, dass eine "offensichtliche Verbesserung" oft wesentlich kleinere Konsequenzen hat, als man zunächst annimmt, und dass sich in manchen Fällen (die wirklich vorkommen) die "Verbesserung" als Verschlechterung herausstellen k-n-

Das Sortieren durch Einfügen scheint also keine sehr geeignete Methode für digitale Rechenanlagen zu sein: Einfügen eines Elementes mit dem nachfolgenden Verschieben einer ganzen Zeile von Elementen um eine einzige Position ist eher unwirtschaftlich. Man sollte bessere Resultate von einer Methode erwarten, die nur einzelne Elemente über grössere Distanzen verschiebt. Diese Idee führt zum Sortieren durch Auswählen.

2.2.2. Sortieren durch direktes Auswählen

Diese Methode beruht auf folgenden Prinzipien:

1. Auswahl des Elementes mit dem kleinsten Schlüssel,

2. Austausch gegen das erste Element a[1] .

Darauf wiederholt man diese Operationen zunächst mit den restlichen n-1 Elementen, dann mit n-2 Elementen, bis schliesslich ein Element - das grösste - übrigbleibt. Diese Methode wird in Tabelle 2.2 mit den gleichen acht Schlüsseln wie in Tabelle 2.1 gezeigt.

ursprüngliche Schlüssel	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

Tabelle 2.2

Das Programm lautet wie folgt:

```
for i := 1 to n-1 do
   begin "weise x das kleinste Element von a[i] ... a[n] zu ,
          weise seinen Index k zu":
         "vertausche a[i] und a[k]"
   end
```

Diese Methode, die direkte Auswahl, ist gewissermassen das Gegenteil von direktem Einfügen: Direktes Einfügen betrachtet bei jedem Schritt nur das eine nächste Element der Quellen-Sequenz und alle Elemente des Ziel-Array, um die Einschubstelle zu finden, während direkte Auswahl alle Elemente des Quellen-Array betrachtet, um das Element mit dem kleinsten Schlüssel zu finden und es als das eine nächste Element der Ziel-Sequenz anzufügen. Das vollständige Programm für direkte Auswahl ist in Programm 2.3 abgebildet.

```
procedure straightselection;
   var i,j,k: index;  x: item;
begin for i := 1 to n-1 do
      begin k := i;  x := a[i];
         for j := i+1 to n do
```

```
            if a[j].key < x.key then
            begin k := j;  x := a[j]
            end ;
        a[k] := a[i]; a[i] := x
    end
end
```

Programm 2.3

Analyse der direkten Auswahl: Die Zahl C der Vergleiche der Schlüssel ist natürlich unabhängig von der ursprünglichen Ordnung der Schlüssel. In diesem Sinn verhält sich diese Methode weniger natürlich als das direkte Einfügen. Wir erhalten

$$C = (n^2 - n)/2$$

Die Zahl M der Bewegungen ist mindestens

$$M[min] = 3*(n-1) \qquad (2.6)$$

bei ursprünglich geordneten Schlüsseln und höchstens

$$M[max] = trunc(n^2 / 4) + 3*(n-1) ,$$

falls die Schlüssel ursprünglich in umgekehrter Reihenfolge sind. Die Berechnung des Mittels M[mit] ist trotz der Einfachheit des Algorithmus kompliziert. Es ist davon abhängig, wie oft beim Durchsuchen einer Sequenz von Zahlen k[1]...k[n] ein k[j] gefunden wird, das kleiner ist als alle vorangehenden Zahlen k[1]...k[j-1]. Dieser Wert, gemittelt über alle n! Permutationen von n Schlüsseln ist H[n]-1, wobei H[n] die n-te harmonische Zahl ist.

$$H[n] = 1 + 1/2 + 1/3 + \ldots + 1/n \qquad (2.7)$$

(vgl. [17], Seite 95-99).

H[n] kann ausgedrückt werden durch

$$H[n] = \ln n + g + 1/(2*n) - 1/(12*n^2) + \ldots \qquad (2.8)$$

mit der Eulerschen Konstante g = 0.577216... . Für hinreichend grosses n können wir die gebrochenen Summanden ignorieren und somit das Mittel der Zuweisungen im i-ten Durchlauf approximieren durch

$$F[i] = \ln i + g + 1 .$$

Die mittlere Anzahl M[mit] der Bewegungen beim Sortieren durch Auswahl ist dann die Summe der F[i] mit i von 1 bis n .

$$M[mit] = \sum_{i=1}^{n} F[i] = n*(g+1) + \sum_{i=1}^{n} \ln i$$

Durch weitere Approximation der Summe mit diskreten Summanden durch das Integral

$$\int_{1}^{n} \ln x \, dx = x*(\ln x - 1) \Big|_{1}^{n} = n*\ln n - n + 1$$

erhalten wir den approximierten Wert

$$M[mit] \approx n*(\ln n + g) \qquad (2.9)$$

Wir können daraus folgern, dass im allgemeinen der Algorithmus der direkten Auswahl dem des direkten Einfügens vorzuziehen ist, obwohl bei anfangs sortierten oder beinahe sortierten Schlüsseln das direkte Einfügen immer noch etwas schneller geht.

2.2.3. Sortieren durch direktes Austauschen

Die Klassifikation von Sortiermethoden ist selten völlig eindeutig. Beide bisher diskutierten Methoden können auch als Sortieren durch Austausch betrachtet werden. In diesem Abschnitt jedoch legen wir eine Methode dar, in der Austausch von zwei Elementen die wesentliche Charakteristik des Prozesses ist. Der nachfolgende Algorithmus der direkten Auswahl beruht auf dem Prinzip des fortgesetzten Vergleichens und Austauschens von Paaren nebeneinanderliegender Elemente, bis alle Elemente sortiert sind.

Wie bei den früheren Methoden der direkten Auswahl durchlaufen wir den Array mehrmals und lassen jedesmal das kleinste Element der restlichen Menge zum linken Ende des Array wandern. Wenn wir zur Abwechslung den Array vertikal statt horizontal betrachten und - mit einiger Phantasie - die Elemente als Blasen (bubbles) auffassen, so steigt bei jedem Durchlauf durch den Array eine Blase auf die ihrem Gewicht (Schlüsselwert) entsprechende Stufe (vgl. Tabelle 2.3).

ursprüngliche Schlüssel	i=2	i=3	i=4	i=5	i=6	i=7	i=8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	62	94	94	94	94	94

Tabelle 2.3

Diese Methode ist bekannt als Bubblesort. Ihre einfachste Form zeigt Programm 2.4:

```
procedure bubblesort;
   var i,j: index;  x: item;
begin for i := 2 to n do
      begin for j := n downto i do
            if a[j-1].key > a[j].key then
            begin x := a[j-1]; a[j-1] := a[j]; a[j] := x
            end
      end
end {bubblesort}
```

Programm 2.4

Dieser Algorithmus lässt sich ohne viel Aufwand verbessern. Das Beispiel in Tabelle 2.3 zeigt, dass die letzten drei Durchgänge

keine Veränderung der Reihenfolge der Elemente bewirken, da die Elemente bereits geordnet sind. Offensichtlich kann dieser Algorithmus verbessert werden, wenn man sich merkt, ob während eines Durchlaufs überhaupt ein Austausch stattgefunden hat. Ein letzter Durchlauf ohne weitere Austauschoperation ist also notwendig, um Termination des Algorithmus zu bestimmen. Diese Verbesserung kann sogar selbst noch verbessert werden, indem man sich nicht nur merkt, dass ein Austausch stattgefunden hat, sondern indem man sich die Position (Index) des letzten Austausches merkt. Denn es ist klar, dass alle Paare nebeneinanderliegender Elemente unterhalb des Index k in der gewünschten Reihenfolge sind. Weitere Durchläufe können daher bei diesem Index abbrechen und müssen nicht bis zu einer vorher bestimmten unteren Grenze i ausgeführt werden. Der aufmerksame Leser wird jedoch eine besondere Asymmetrie feststellen: Eine einzige falsch plazierte Blase im "schweren" Ende eines sonst sortierten Array wird in einem einzigen Durchlauf an ihren Platz gebracht, während ein im "leichten" Ende falsch plaziertes Element bei jedem Durchlauf nur um einen Schritt zu seinem richtigen Platz vorrückt. Zum Beispiel wird der Array

12 18 42 44 55 67 94 06

durch den verbesserten Bubblesort in einem Durchlauf sortiert, während der Array

94 06 12 18 42 44 55 67

zum Sortieren sieben Durchläufe benötigt. Diese unnatürliche Asymmetrie legt eine dritte Verbesserung nahe: die Aenderung der Richtung aufeinanderfolgender Durchläufe. Entsprechend nennen wir diesen Algorithmus Shakersort. Tabelle 2.4 zeigt sein Verhalten bei Anwendung auf die gleichen acht Schlüssel wie in Tabelle 2.3.

```
procedure shakersort;
   var j,k,l,r: index;  x:item;
begin l := 2; r := n; k := n;
   repeat
      for j := r downto l do
         if a[j-1].key > a[j].key then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
```

```
            k := j
         end ;
      l := k+1;
      for j := l to r do
         if a[j-1].key > a[j].key then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end ;
      r := k-1
   until l > r
end {shakersort}
```

Programm 2.5

l=2	3	3	4	4
r=8	8	7	7	4
44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	42	44
18	94	18	55	55
06	18	67	67	67
67	67	94	94	94

Tabelle 2.4

Analyse von Bubblesort und Shakersort: Die Zahl der Vergleiche im direkten Austausch-Algorithmus ist

$$C = (n^{**}2 - n)/2 \qquad (2.10)$$

und die minimale, mittlere und maximale Zahl der Bewegungen (Zuweisungen von Elementen) sind

$$M[min] = 0 \qquad (2.11)$$
$$M[mit] = 3*(n^{**}2 - n)/4$$
$$M[max] = 3*(n^{**}2 - n)/2$$

Die Analyse der verbesserten Methoden, besonders des Shakersort, ist ziemlich kompliziert. Die kleinste Zahl der Vergleiche ist C[min] = n-1. Für den verbesserten Bubblesort berechnet Knuth eine mittlere Zahl von Durchläufen proportional zu n - k1*sqrt(n) und eine mittlere Zahl von Vergleichen proportional zu (n**2-n(k2 + ln(n)))/2. Zu beachten ist, dass alle oben angeführten Verbesserungen die Zahl der Austausche nicht verändern; sie vermindern lediglich die Zahl der unnötigen mehrfachen Tests. Leider ist ein Austausch von zwei Elementen im allgemeinen eine sehr viel aufwendigere Operation als ein Vergleich von zwei Schlüsseln; unsere trickreichen Verbesserungen haben daher keine so tiefgreifende Wirkung, wie man intuitiv erwarten würde.

Die Analyse zeigt, dass das Sortieren durch Austausch und seine kleinen Verbesserungen sowohl dem Sortieren durch Einfügen als auch durch Auswahl unterlegen sind; tatsächlich enthält Bubblesort wenig Empfehlenswertes ausser einem gefälligen Namen. Der Algorithmus Shakersort ist in den Fällen vorteilhaft, in denen die Elemente bereits weitgehend geordnet sind - ein in der Praxis seltener Fall.

Man kann zeigen, dass jedes der n Elemente während des Sortierens eine mittlere Distanz von n/3 Plätzen zurückzulegen hat. Diese Zahl ist ein Anhaltspunkt für die Suche nach besseren, d.h. effektiveren Sortiermethoden. Alle direkten Sortiermethoden bewegen im wesentlichen jedes Element um eine Position bei jedem elementaren Schritt. Deshalb benötigen sie eine Anzahl von O(n**2) Schritten. Jede Verbesserung muss also auf dem Prinzip beruhen, Elemente über grössere Distanzen in einzelnen Sprüngen zu bewegen.

Nachfolgend werden drei verbesserte Methoden untersucht, je eine für jede der elementaren Sortiermethoden: Einfügen, Auswahl und Austausch.

2.2.4. Sortieren durch Einfügen mit abnehmender Schrittweite

Eine Verfeinerung des Sortierens durch direktes Einfügen wurde 1959 von D.L. Shell vorgeschlagen. Die Methode wird erklärt und vorgeführt an unserem Standard-Beispiel der acht Elemente (vgl. Tabelle 2.5). Zuerst werden alle Elemente, die vier Positionen

entfernt sind, zusammengefasst und getrennt sortiert. Dieser Prozess heisst 4-Sortierung. In dem Beispiel der acht Elemente enthält jede Gruppe genau zwei Elemente. Nach diesem ersten Durchlauf werden die Elemente erneut in Gruppen von Elementen zusammengefasst, die zwei Positionen entfernt sind, und dann erneut sortiert. Dieser Prozess heisst 2-Sortierung. Schliesslich werden in einem dritten Durchlauf alle Elemente in einer gewöhnlichen Sortierung oder einer 1-Sortierung geordnet.

44 55 12 42 94 18 06 67

4-Sortierung ergibt

44 18 06 42 94 55 12 67

2-Sortierung ergibt

06 18 12 42 44 55 94 67

1-Sortierung ergibt

06 12 18 42 44 55 67 94

Tabelle 2.5

Man wird sich zunächst wundern, ob die Notwendigkeit mehrerer Durchläufe zum Sortieren, wobei jeweils alle Elemente betroffen sind, nicht grösseren Aufwand verursacht als erspart. Aber jeder Schritt des Sortierens einer Kette betrifft entweder relativ wenige Elemente, oder die Elemente sind bereits sehr gut sortiert, so dass verhältnismässig wenige Umstellungen erforderlich sind.

Es ist klar, dass die Methode einen geordneten Array liefert, und es ist ziemlich klar, dass jeder Durchlauf aus den vorhergehenden Durchläufen Nutzen zieht (da jede i-Sortierung zwei durch die vorangehende 2i-Sortierung geordnete Gruppen zusammenfasst). Es ist ausserdem klar, dass jede Folge von Schrittweiten verwendet werden kann, solange nur die letzte Schrittweite 1 beträgt, da im schlimmsten Fall der letzte Durchlauf die ganze Arbeit verrichtet. Es ist dagegen viel weniger klar, dass die Methode der kleiner wer-

denden Schrittweiten sogar bessere Resultate liefert mit Schrittweiten, die keine Potenzen von 2 sind.

Das Programm wird deshalb entwickelt, ohne sich auf eine spezielle Folge der Schrittweiten zu beziehen. Die t Schrittweiten werden bezeichnet durch

$$h[1],\ h[2],\ \ldots\ ,\ h[t]$$

mit der Bedingung, dass

$$h[t] = 1,\ h[i+1] < h[i] \qquad (2.12)$$

Jede h-Sortierung wird programmiert als ein Sortieren mit direktem Einfügen unter Verwendung einer Marke, um ein einfaches Abbruchkriterium bei der Suche der Einschubstelle zu erhalten.

Jedes Sortieren muss natürlich seine eigene Marke setzen, und das Programm zur Bestimmung der Position soll so einfach wie möglich sein. Der Array a muss deshalb nicht nur um eine einzelne Komponente a[0], sondern um h[1] Komponenten erweitert werden, so dass er jetzt vereinbart wird als

```
a: array [-h1..n] of item .
```

Der Algorithmus wird in der folgenden Prozedur Shellsort [26] beschrieben (vgl. Programm 2.6).

```
procedure shellsort;
   const t = 4;
   var i,j,k,s: index;  x: item;  m: 1..t;
      h: array [1..t] of integer;
begin h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
   for m := 1 to t do
   begin k := h[m]; s := -k;  {Position der Marke}
      for i := k+1 to n do
      begin x := a[i]; j := i-k;
         if s=0 then s := -k; s := s+1; a[s] := x;
         while x.key < a[j].key do
           begin a[j+k] := a[j]; j := j-k
```

```
                end;
              a[j+k] := x
            end
        end
    end
```

Programm 2.6

Analyse des Shellsort: Die Analyse dieses Algorithmus stellt einige sehr schwierige mathematische Probleme, von denen viele bis jetzt noch ungelöst sind. Insbesondere ist noch nicht bekannt, welche Wahl der Schrittweiten die besten Resultate liefert. Ueberraschend ist aber, dass sie nicht Vielfache voneinander sein sollen. Dies verhindert die aus obigem Beispiel ersichtliche Erscheinung, dass jeder Durchlauf beim Sortieren zwei Ketten kombiniert, die vorher überhaupt keine Verbindung hatten. Es ist sogar wünschenswert, dass so oft wie möglich Verbindungen zwischen verschiedenen Ketten stattfinden, und es gilt das folgende Theorem:

> Wenn eine k-sortierte Sequenz i-sortiert wird, bleibt sie k-sortiert.

Knuth [18] gibt Anhaltspunkte, nach denen die (in umgekehrter Reihenfolge geschriebene) Folge von Schrittweiten

1, 4, 13, 40, 121, ...

mit h[k-1] = 3*h[k]+1 , h[t] = 1 und t = trunc(log3 n)-1 eine vernünftige Wahl darstellt. Er empfiehlt ausserdem die Sequenz

1, 3, 7, 15, 31, ...

mit h[k-1] = 2*h[k]+1 , h[t] = 1 und t = trunc(log2 n) - 1. Für die letztere Wahl ergibt eine mathematische Analyse für das Sortieren von n Elementen mit dem Algorithmus Shellsort einen zu n**1.2 proportionalen Aufwand. Obwohl dies eine wesentliche Verbesserung gegenüber n**2 ist, wollen wir diese Methode nicht weiterverfolgen, da es noch bessere Algorithmen gibt.

2.2.5. Sortieren mit Bäumen

Die Methode des Sortierens durch direkte Auswahl beruht auf dem wiederholten Auslesen des kleinsten Schlüssels aus n Elementen, aus den restlichen n-1 Elementen, usw. Natürlich benötigt das Finden des kleinsten Schlüssels unter n Elementen n-1 Vergleiche, das Finden unter n-1 Elementen benötigt n-2 Vergleiche, usw. Wie kann nun das Sortieren durch Auswahl verbessert werden? Es lässt sich nur verbessern, wenn man von jedem Durchlauf mehr Information behält als nur die Identifikation des kleinsten Elementes. Zum Beispiel kann mit n/2 Vergleichen der kleinere Schlüssel von jedem Paar von Elementen bestimmt werden, mit weiteren n/4 Vergleichen kann der kleinere von jedem Paar solcher kleinerer Schlüssel gewählt werden, usw. Schliesslich können wir mit nur n-1 Vergleichen den Auswahlbaum von Fig. 2.3 konstruieren und die Wurzel als den gewünschten kleinsten Schlüssel identifizieren [7].

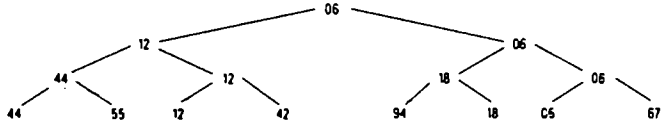


Fig 2.3

Der zweite Schritt besteht nun aus dem Hinabsteigen entlang dem durch den kleinsten Schlüssel markierten Weg, wobei dieser sukzessiv bis zum Ende entweder durch ein leeres Element ersetzt wird oder durch das Element des anderen Zweiges bei dazwischenliegenden Knoten (vgl. Fig. 2.4 und 2.5).

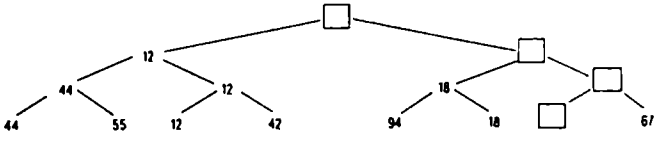


Fig. 2.4

Wiederum hat das nun an der Wurzel erscheinende Element des Baumes den (nun zweit-) kleinsten Schlüssel und kann entfernt werden. Nach n solchen Auswahlschritten ist der Baum leer (d.h. voller Löcher),

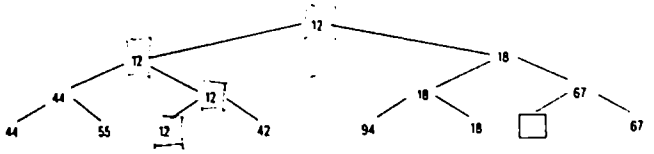


Fig. 2.5

und der Sortierprozess ist beendet. Man beachte, dass jeder der n Auswahlschritte nur log2(n) Vergleiche benötigt. Der vollständige Auswahlprozess braucht deshalb nur O(n*log(n)) elementare Operationen zusätzlich zu den n-1 Schritten, die zur Erstellung des Baumes nötig waren. Dies ist eine sehr wesentliche Verbesserung gegenüber den direkten Methoden mit O(n**2) Schritten und sogar gegenüber Shellsort mit O(n**1.2) Schritten.

Natürlich wird die Buchführung aufwendiger, und deshalb ist die Komplexität der einzelnen Schritte bei der Methode des Sortierens mit Bäumen grösser; schliesslich muss zum Aufbewahren der grösseren Menge von Information aus dem einleitenden Durchlauf eine Art Baumstruktur aufgebaut werden. Als nächstes müssen wir Methoden suchen, mit denen wir diese Information effizient organisieren können.

Es wäre natürlich äusserst wünschenswert, die Notwendigkeit für die Löcher zu eliminieren, die am Ende den ganzen Baum bevölkern und die Anlass zu vielen unnötigen Vergleichen geben. Ausserdem sollte eine Darstellung des Baumes mit n Elementen in n Speichereinheiten gefunden werden statt in 2n-1 Einheiten, wie es oben gezeigt wurde. Diese Ziele werden durch den von seinem Schöpfer J. Williams sogenannten Heapsort erreicht [30]; diese Methode stellt tatsächlich eine drastische Verbesserung gegenüber konventionellen Versuchen des Sortierens mit Bäumen dar.

Ein Heap (= Haufen) ist definiert als eine Folge von Schlüsseln

$$h[l], h[l+1], \ldots, h[r],$$

mit den Relationen

$$h[i] <= h[2*i] \qquad (2.13)$$
$$h[i] <= h[2*i+1]$$

für alle i = l ... r/2 . Wenn ein binärer Baum wie in Fig. 2.6 als

Array dargestellt wird, folgt,

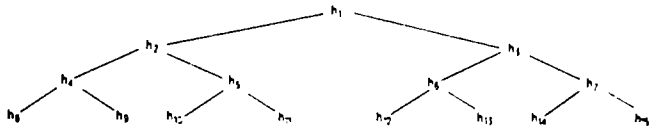


Fig. 2.6

dass die Sortierbäume in Fig. 2.7 und 2.8 Heaps sind, und insbesondere, dass das Element h[1] das kleinste Element des Heap ist.

$$h[1] = \min(h[1] \ldots h[n])$$

Wir wollen nun annehmen, dass ein Heap mit Elementen h[l+1]...h[r] für beliebige Werte l und r gegeben ist und dass ein neues Element x hinzugefügt werden soll, wodurch sich ein erweiterter Heap h[l]...h[r] ergibt. Gehen wir z.B. vom Heap h[2]...h[7] in Fig. 2.7 aus, und erweitern wir den Heap "nach links" um ein Element

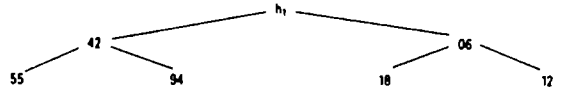


Fig. 2.7

h[1] = 44. Wir erhalten einen neuen Heap, indem wir x zunächst auf die Spitze der Baumstruktur setzen und es dann "nach unten sinken" lassen entlang dem Weg der kleineren verglichenen Elemente, die gleichzeitig nach oben wandern. Bei diesem Beispiel wird der Wert 44 zuerst ausgetauscht gegen 06, dann gegen 12, und es ergibt sich der Baum von Fig. 2.8.

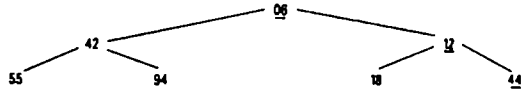


Fig. 2.8

Wir formulieren nun diesen Algorithmus des Einfügens folgendermassen: i,j sind das Paar der Indizes, das die in jedem Schritt auszutauschenden Elemente bezeichnet. Der Leser soll sich selbst

überzeugen, dass die vorgeschlagene Methode des Einfügens alle Bedingungen unverändert lässt, durch die ein Heap definiert ist. (to sift = sieben, sondern)

```
procedure sift(l,r: index);
   label 99;
   var i,j: index; x: item;
begin i := l; j := 2*i; x := a[i];
   while j <= r do
   begin if j < r then
            if a[j].key > a[j+1].key then j := j+1;
         if x.key <= a[j].key then goto 99;
         a[i] := a[j]; i := j; j := 2*i  {einfügen}
   end;
99: a[i] := x
end
```

Programm 2.7

Eine geschickte Art der Konstruktion eines Heap am Ort wurde von R.W. Floyd vorgeschlagen und verwendet die in Programm 2.7 gezeigte Prozedur zum Einfügen. Gegeben ist ein Array h[1] ... h[n]; die Elemente h[⌊n/2⌋+1] .... h[n] bilden bereits einen Heap, da keine zwei Indizes i,j mit j = 2i (oder j = 2i+1) existieren. Diese Elemente können wir als unterste Zeile des zugeordneten binären Baumes auffassen (vgl. Fig. 2.6); zwischen ihnen wird ja keine Ordnungsrelation gefordert. Der Heap wird nun nach links erweitert, wobei in jedem Schritt ein neues Element hinzukommt und an der richtigen Stelle eingefügt wird. Dieser Prozess wird in Tabelle 2.6 illustriert und liefert den Heap von Fig. 2.6 .

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
44	55	06	42	94	18	12	67
44	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

Tabelle 2.6

Folglich lässt sich der Prozess der Erstellung eines Heap mit n

Elementen h[1] ... h[n] am Ort beschreiben durch

```
l := (n div 2) + 1;
while l > 1 do
   begin l := l-1; sift(l,n)
   end
```

Zum Sortieren der Elemente sind nun n Einfügungsschritte erforderlich, wobei nach jedem Schritt das folgende Element an der Spitze des Heap weggenommen werden kann. Einmal mehr stellt sich die Frage, wo die sich an der Spitze ergebenden Elemente zu speichern sind und ob Sortieren am Ort möglich ist. Natürlich gibt es eine solche Lösung! Bei jedem Schritt nehme man die letzte Komponente, sagen wir x, aus dem Heap, speichere das Element an der Spitze des Heap an dem nun freien Platz von x und lasse x an seine richtige Position nach unten gleiten. Der Prozess wird mit Hilfe der Prozedur sift (Programm 2.7) folgendermassen beschrieben:

```
r := n;
while r > 1 do
   begin x := a[1]; a[1] := a[r]; a[r] := x
      r := r-1; sift(1,r)
   end
```

Die notwendigen n-1 Schritte sind aus Tabelle 2.7 ersichtlich.

06	42	12	55	94	18	44	67 |
12	42	18	55	94	67	44 |	06
18	42	44	55	94	67 |	12	06
42	55	44	67	94 |	18	12	06
44	55	94	67 |	42	18	12	06
55	67	94 |	44	42	18	12	06
67	94 |	55	44	42	18	12	06
94 |	67	55	44	42	18	12	06

Tabelle 2.7

Das Beispiel von Tabelle 2.7 zeigt, dass die resultierende Ordnung gerade umgekehrt ist. Dies lässt sich aber leicht ändern, indem man die Ordnungsrelationen in der Prozedur für das Einfügen ändert.

Dies ergibt die in Programm 2.8 dargelegte Prozedur Heapsort.

```
procedure heapsort;
   var l,r: index;  x: item;

   procedure sift;
      label 13;
      var i,j: index;
   begin i := l; j := 2*i; x := a[i];
      while j <= r do
      begin if j < r then
               if a[j].key < a[j+1].key then j := j+1;
         if x.key >= a[j].key then goto 13;
         a[i] := a[j]; i := j; j := 2*i
      end ;
   13: a[i] := x
   end ;

begin l := (n div 2) + 1; r := n;
   while l > 1 do
      begin l := l-1; sift
      end ;
   while r > 1 do
      begin x := a[1]; a[1] := a[r]; a[r] := x;
         r := r-1; sift
      end
end {heapsort}
```

Programm 2.8

Analyse des Heapsort: Auf den ersten Blick ist es nicht offensichtlich, dass diese Sortiermethode gute Resultate liefert. Schliesslich werden grosse Elemente nach links verschoben, bevor sie endlich ganz rechts landen. Tatsächlich kann diese Prozedur für kleine Anzahlen von Elementen wie in dem gezeigten Beispiel nicht empfohlen werden. Für grosse n aber ist Heapsort sehr effizient, und je grösser n, um so besser ist er sogar im Vergleich zu Shellsort.

Schlimmstenfalls sind n/2 Schritte für das Einfügen von Elementen

in log(n/2), log(n/2-1), ... , log(n-1) Positionen nötig, wobei der Logarithmus zur Basis 2 zu nehmen und auf die nächste kleinere, ganze Zahl abzuschneiden ist. Folglich benötigt die Sortierphase n-1 Einfügeschritte mit höchstens log(n-1), log(n-2), ... , 1 Bewegungen. Dazu kommen noch n-1 Bewegungen für das nach rechts Schieben des geordneten Elementes. Dieses Argument zeigt, dass Heapsort auch im schlimmsten Fall O(n*log(n)) Schritte benötigt. Diese hervorragende Leistung im schlimmsten Fall ist eine der stärksten Eigenschaften des Heapsort.

Es ist durchaus nicht klar, in welchem Fall die schlechteste (oder die beste) Leistung zu erwarten ist. Aber im allgemeinen scheinen Heapsort Sequenzen zu liegen, die von Anfang an mehr oder weniger in der umgekehrten Reihenfolge geordnet sind, und er zeigt damit ein unnatürliches Verhalten. Das Aufbauen des Heap erfordert keine Bewegungen, wenn eine umgekehrte Reihenfolge vorliegt. Für die acht Elemente unseres Beispiels werden für folgende Ausgangs-Sequenzen die minimale und maximale Anzahl von Bewegungen benötigt:

```
M[min]  = 13   für die Sequenz
   94   67   44   55   12   42   18   06
M[max]  = 24   für die Sequenz
   18   42   12   44   06   55   67   94
```

Die mittlere Zahl der Bewegungen ist ungefähr (n*log n)/2, und die Abweichungen von diesem Wert sind relativ klein.

2.2.6. Sortieren durch Zerlegen (Partition)

Nach der Erörterung der beiden höheren Sortiermethoden, die auf dem Prinzip des Einfügens und Auswählens aufbauen, führen wir nun eine dritte, verbesserte Methode ein, die auf dem Prinzip des Austauschens beruht. Unter Berücksichtigung der Tatsache, dass Bubblesort im Mittel die schlechteste Effizienz unter den drei direkten Sortier-Algorithmen hatte, sollte man einen signifikanten Verbesserungsfaktor erwarten. Dennoch ist es überraschend, dass die hier erörterte Verbesserung, die auf Austauschen beruht, die beste bisher bekannte Sortiermethode mit Arrays liefert. Ihre Leistung ist so beeindruckend, dass ihr Erfinder C.A.R. Hoare sie Quicksort

[9,12] nannte.

Quicksort stützt sich auf die Tatsache, dass Austauschen vorzugsweise über grössere Distanzen ausgeführt wird und dann am effizientesten ist. Sind z.B. n Elemente in umgekehrter Reihenfolge ihrer Schlüssel gegeben, so kann man sie mit nur n/2 Austauschvorgängen sortieren, indem man, links und rechts aussen beginnend, schrittweise von beiden Seiten nach innen vorgeht. Natürlich ist dies nur möglich, wenn wir wissen, dass die Reihenfolge gerade umgekehrt ist. Aber aus diesem Beispiel kann man dennoch etwas lernen.

Versuchen wir den folgenden Algorithmus: Man wähle willkürlich ein Element (und nenne es x), durchsuche dann den Array von links, bis ein Element a[i] > x gefunden wird, und von rechts, bis ein Element a[j] < x gefunden wird. Nun sind diese beiden Elemente zu vertauschen, und dieser Prozess des "Durchsuchens und Vertauschens" ist so lange fortzusetzen, bis man sich beim Durchsuchen aus beiden Richtungen irgendwo trifft. Als Resultat ist jetzt der Array zerlegt in einen linken Teil mit Schlüsseln kleiner als x und einen rechten Teil mit Schlüsseln grösser als x. Dieser Zerlegungsprozess ist formuliert als Prozedur in Programm 2.9. (Man beachte, dass die Relationen > und < ersetzt wurden durch >= und <=, deren Negationen < und > in der while-Klausel vorkommen. Damit wird x zu einer Marke für das Durchsuchen.)

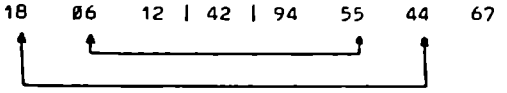
```
procedure partition;
   var w,x: item;
begin i := 1; j := n;
   "wähle ein willkürliches Element x = a[k]";
   repeat
      while a[i].key < x.key do i := i+1;
      while x.key < a[j].key do j := j-1;
      if i <= j then
         begin w := a[i]; a[i] := a[j]; a[j] := w;
               i := i+1; j := j-1
         end
   until i > j
end
```

Programm 2.9

Wird z.B. der mittlere Schlüssel 42 als Vergleichswert x gewählt, so erfordert der Array von Schlüsselwerten

44 55 12 42 94 06 18 67

für folgende Zerlegung des Array zwei Vertauschungen



mit den Endwerten der Indizes i = 5 und j = 3. Die Schlüssel a[1]...a[j-1] sind kleiner oder gleich dem Schlüssel von x (= 42), die Schlüssel von a[i+1]...a[n] sind grösser oder gleich dem Schlüssel von x. Folglich gibt es zwei Teile, nämlich

```
a[k].key <= x.key     für     k = 1...i-1          (2.14)
a[k].key >= x.key     für     k = j+1...n
```

und damit

```
a[k].key = x.key      für     k = i-1...j+1        (2.15)
```

Dieser Algorithmus ist offensichtlich sehr direkt und effizient, da die wesentlichen zu vergleichenden Werte i, j und x während des Durchsuchens in schnellen Registern bleiben können. Er kann aber auch schwerfällig sein, wie der Fall mit n gleichen Schlüsseln zeigt, in dem n/2 Austausche vorgenommen werden. Dieses unnötige Austauschen ist leicht zu eliminieren, wenn man die Anweisungen für das Durchsuchen abändert zu

```
while a[i].key <= x.key do i := i+1;
while x.key <= a[j].key do j := j-1;
```

In diesem Fall dient aber das ausgewählte Element x, das selbst im Array liegt, nicht mehr als Marke für die beiden Durchläufe. Der Array mit identischen Schlüsseln würde bewirken, dass das Durchlaufen über die Grenzen des Array hinausgeht, wenn nicht kompliziertere Abbruchbedingungen verwendet würden. Für die Einfachheit der Bedingungen in Programm 2.9 nimmt man die zusätzlichen Austausche

in Kauf, die im mittleren "zufälligen" Fall realtiv selten vorkommen. Eine leichte Ersparnis kann aber erreicht werden, wenn man die Kontrollklausel für den Austauschschritt von i <= j ändert zu

```
i < j .
```

Diese Aenderung darf aber nicht ausgedehnt werden auf die beiden Anweisungen

```
i := i+1;  j := j-1
```

die somit eine getrennte Bedingungsklausel benötigen. Das folgende Beispiel mit x = 2 zeigt die Notwendigkeit dieser Klausel:

```
1   1   1   2   1   1   1
```

Der erste Durchlauf und Austausch ergibt

```
1   1   1   1   1   1   2
```

und i = 5, j = 6. Beim zweiten Durchlauf bleibt der Array unverändert mit i = 7, j = 6. Wäre der Austausch nicht durch die Bedingung i <= j unterdrückt worden, so hätte ein falscher Austausch von a[6] mit a[7] stattgefunden.

Vertrauen in die Richtigkeit des Zerlegungs-Algorithmus kann man gewinnen, indem man verifiziert, dass die beiden Zusicherungen (2.14) Invarianten der repeat-Anweisung sind. Zu Beginn, für i = 1 und j = n, sind sie trivialerweise richtig, und am Ende, für i > j, implizieren sie das gewünschte Resultat.

Wir müssen uns nun daran erinnern, dass unser Ziel nicht nur das Finden von Zerlegungen des ursprünglichen Array von Elementen ist, sondern seine Sortierung. Es ist jedoch nur ein kleiner Schritt vom Zerlegen zum Sortieren: Nach dem Zerlegen das Array wende man den gleichen Prozess auf beide Teile an, dann auf die Teile dieser Zerlegung, usw., bis jeder Teil nur ein Element umfasst. Programm 2.10 beschreibt dieses Rezept.

```
procedure quicksort;
   procedure sort(l,r: index);
      var i,j: index; x,w: item;
   begin i := l; j := r;
      x := a[(l+r) div 2];
      repeat
         while a[i].key < x.key do i := i+1;
         while x.key < a[j].key do j := j-1;
         if i <= j then
         begin w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1
         end
      until i > j;
      if l < j then sort(l,j);
      if i < r then sort(i,r)
   end;
begin sort(1,n)
end (quicksort)
```

Programm 2.10

Die Prozedur sort ruft sich selbst rekursiv auf. Diese Rekursion in Algorithmen ist ein sehr wirkungsvolles Werkzeug und wird in Kapitel 3 ausführlich behandelt. In einigen Programmiersprachen älteren Datums wird Rekursion aus gewissen technischen Gründen nicht zugelassen. Wir werden nun zeigen, wie dieser gleiche Algorithmus auch als nicht rekursive Prozedur ausgedrückt werden kann. Offensichtlich liegt die Lösung im Ersetzen der Rekursion durch Iteration, wobei gewisse zusätzliche Operationen zur Buchhaltung nötig werden.

Der Schlüssel zu einer iterativen Lösung liegt im Führen einer Liste von notwendigen Zerlegungen, die noch auszuführen sind. Nach jedem Schritt ergibt sich die Forderung nach zwei weiteren Zerlegungen. Nur eine davon kann in der anschliessenden Iteration direkt in Angriff genommen werden, die andere wird in dieser Liste gespeichert. Es ist natürlich wesentlich, dass diese Liste in einer bestimmten, nämlich der umgekehrten, Reihenfolge abgearbeitet wird. Dies bedeutet, dass die erste eingetragene Zerlegung als letzte ausgeführt wird und umgekehrt; die Liste wird wie ein pulsierender

Stapel (Stack) behandelt. In der folgenden nicht rekursiven Version des Quicksort wird jede verlangte Zerlegung einfach durch einen linken und einen rechten Index dargestellt, die die Grenzen des weiterzuzerlegenden Teils bilden. Dazu führen wir eine Array-Variable stack und einen Index s ein, der die letzte Eintragung angibt (vgl. Programm 2.11). Die geeignete Wahl der Stack-Länge M wird in der nachfolgenden Analyse des Quicksort erörtert.

```
procedure quicksort1;
   const m = 12;
   var i,j,l,r: index;
      x,w: item;
      s: 0 .. m;
      stack: array [1..m] of
               record l,r: index end ;

begin s := 1; stack[1].l := 1; stack[1].r := n;
   repeat {nimm oberste Anweisung vom Stack}
      l := stack[s].l; r := stack[s].r; s := s-1;
      repeat {teile a[l] ... a[r] auf}
         i := l; j := r; x := a[(l+r) div 2];
         repeat
            while a[i].key < x.key do i := i+1;
            while x.key < a[j].key do j := j-1;
            if i <= j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
               i := i+1; j := j-1
            end
         until i > j;
         if i < r then
         begin {speichere Anweisung für Sortieren des rechten
                Teils auf dem Stack}
            s := s+1; stack[s].l := i; stack[s].r := r
         end ;
         r := j
      until l >= r
   until s = 0
end {quicksort1}
```

Programm 2.11

Analyse des Quicksort: Um die Leistung des Quicksort zu analysieren, müssen wir zuerst das Verhalten des Zerlegungsprozesses untersuchen. Nach der Wahl einer Grenze x wird der ganze Array abgesucht. Dabei werden genau n Vergleiche ausgeführt. Die Zahl der Austauschoperationen kann durch folgende wahrscheinlichkeitstheoretische Ueberlegung bestimmt werden. Nehmen wir an, die zu zerlegende Datenmenge besteht aus den n Schlüsseln 1 ... n, und wählen wir x als Grenze. Nach der Zerlegung wird x die x-te Position im Array einnehmen. Die Zahl der notwendigen Austauschoperationen ist gleich der Zahl der Elemente im linken Teil der Zerlegung (x-1), multipliziert mit der Wahrscheinlichkeit, dass ein Schlüssel vertauscht wurde. Ein Schlüssel wird ausgetauscht, wenn er nicht kleiner ist als die Grenze x. Diese Wahrscheinlichkeit ist (n-x+1)/n . Die zu erwartende Zahl der Austauschoperationen ergibt sich als die durch n dividierte Summe über alle möglichen Wahlen der Grenze.

$$M = (1/n) \sum_{x=1}^{n} (n-x)*(n-x+1)/n = (n/6) - 1/(6*n)$$

Somit ist die erwartete Zahl der Austauschoperationen ungefähr n/6.

Nehmen wir an, wir hätten viel Glück und würden immer das mittlere Element als Grenze wählen, dann teilt jede Zerlegung den Array in zwei Hälften, und zum Sortieren sind log(n) Durchläufe notwendig. Damit ergeben sich insgesamt n*log(n) Vergleiche und (n*log(n))/6 Austauschoperationen.

Natürlich kann man nicht erwarten, immer das mittlere Element zu treffen. Die Wahrscheinlichkeit dafür ist nur 1/n. Ueberraschenderweise ist aber die durchschnittliche Leistung von Quicksort bei zufälliger Wahl der Grenze nur um einen Faktor 2*ln(2)≈1.39 schlechter als im besten Fall.

Nehmen wir wieder unser Standard-Beispiel mit acht Elementen, so finden wir, dass die Folge

55 94 67 44 12 06 18 42

nur neun Austauschschritte erfordert.

Aber auch Quicksort hat seine Tücken. Zunächst ist seine Leistung für kleine Werte von n relativ schwach, wie es bei allen höheren Methoden der Fall ist. Sein Vorzug gegenüber den anderen Methoden liegt in der Leichtigkeit, mit der eine direkte Sortiermethode zum Sortieren kleiner Zerlegungsteile eingebaut werden kann. Dies ist bei einer rekursiven Version des Programms besonders von Vorteil.

Es bleibt immer noch die Frage des schlimmsten Falls. Wie verhält sich Quicksort dann? Die Antwort ist leider enttäuschend und deckt die eine Schwäche des Quicksort auf (der in diesen Fällen zum Slowsort wird). Betrachtet man z.B. den ungünstigen Fall, bei dem jedesmal der grösste Wert einer Zerlegung als Vergleichswert x gewählt wird. Bei jedem Schritt wird dann ein Segment mit n Elementen in einen linken Teil mit n-1 und einen rechten Teil mit einem einzigen Element zerlegt. Als Folge werden n anstatt nur log(n) Zerlegungen notwendig, und die Leistung im schlimmsten Fall ist somit von der Grössenordnung n**2.

Anscheinend ist der entscheidende Schritt die Wahl des Vergleichswertes x. In unserem Beispielprogramm wird dazu das mittlere Element gewählt. Man beachte, dass man praktisch ebenso gut das erste oder letzte Element a[l] oder a[r] wählen könnte. Dabei wäre der schlimmste Fall der ursprünglich sortierte Array; Quicksort legt dann eine bestimmte Abneigung gegen den trivialen Fall an den Tag und bevorzugt ungeordnete Arrays. Bei der Wahl des mittleren Elementes kommt diese sonderbare Eigenschaft des Quicksort etwas weniger zur Geltung, da der ursprünglich geordnete Array der günstigste Fall ist. Tatsächlich ist die Leistung im Mittel etwas besser, wenn das mittlere Element gewählt wird. Hoare schlägt vor, die Wahl von x "willkürlich" zu treffen oder den mittleren aus einer kleinen Auswahl von etwa drei Schlüsseln zu wählen [27][28]. Solch eine ausgewogene Wahl beeinflusst die durchschnittliche Leistung des Quicksort kaum, sie verbessert aber die Leistung im schlimmsten Fall wesentlich. Es kommt zum Ausdruck, dass Sortieren mit Quicksort einem Glücksspiel ähnelt, bei dem man sich bewusst sein sollte, welchen Verlust man sich leisten kann, wenn man Pech hat.

Es ist eine wichtige Lehre aus dieser Erfahrung zu ziehen: sie betrifft ganz besonders den Programmierer. Welche Folgen hat der

obenerwähnte schlimmste Fall für Programm 2.10? Wir haben bemerkt, dass jede Aufteilung einen rechten Teil mit nur einem Element liefert; die Notwendigkeit, diesen Teil zu sortieren, wird für die spätere Ausführung im Stack gespeichert. Folglich ist n die grösste Zahl solcher Anforderungen und damit auch die benötigte Stack-Länge. Dies ist natürlich völlig unannehmbar. (Man beachte, dass wir mit der rekursiven Version nicht besser - sondern sogar schlechter - fahren, da ein System mit rekursiver Verwendung von Prozeduren die Werte der lokalen Variablen und Parameter für jeden Aufruf der Prozedur automatisch speichern muss und dazu einen impliziten Stack verwendet.) Die Lösung liegt im Speichern der Forderung nach Sortierung für den grösseren Teil der Zerlegung und in der direkten Weiterzerlegung des kleineren Teils. In diesem Fall kann die Stack-Länge M auf M = log2 n beschränkt werden.

Die in Programm Quicksort1 notwendige Aenderung betrifft den Teil, in dem neue Forderungen aufgestellt werden. Er lautet nun

```
if j-l < r-i then                                          (2.16)
begin if i < r then
      begin {Stack-Eintragung für Sortieren des
             rechten Teils}
            s := s+1; stack[s].l := i; stack[s].r := r
      end;
      r := j  {weitersortieren des linken Teils}
end else
begin if l < j then
      begin {Stack-Eintragung für Sortieren des
             linken Teils}
            s := s+1; stack[s].l := l; stack[s].r := j
      end;
      l := i  {weitersortieren des rechten Teils}
end
```

2.2.7. Bestimmung des mittleren Elementes

Das mittlere von n Elementen (median) ist definiert als das Element, das kleiner (oder gleich) ist als die Hälfte der n Elemente und grösser (oder gleich) als die andere Hälfte der

Elemente. Zum Beispiel ist das mittlere von

16 12 99 95 18 87 10

gleich 18.

Das Problem der Bestimmung des mittleren Elementes ist gewöhnlich mit dem des Sortierens verknüpft, da eine sichere Methode hierfür das Sortieren der n Elemente ist, wobei man dann das Element in der Mitte nimmt. Aber Quicksort öffnet einen Weg zu einer möglicherweise viel schnelleren Art der Bestimmung dieses Elementes. Die leicht zu erklärende Methode lässt sich verallgemeinern zu dem Problem der Bestimmung des k-kleinsten von n Elementen. Die Bestimmung des mittleren wird dann zum Sonderfall k = n/2.

Der von C.A.R. Hoare gefundene Algorithmus funktioniert folgendermassen [11]. Zunächst erfolgt eine Zerlegung nach Quicksort mit l = 1 und r = n und mit a[k] als Trennwert x. Für die resultierenden Indexwerte i und j gilt

1. a[h] <= x für alle h < i und (2.17)
2. a[h] >= x für alle h > j und
3. i > j .

Es können drei Fälle eintreten

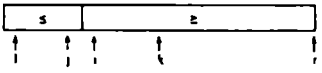


Fig. 2.9

1. Der Trennwert x war zu klein; folglich liegt die Grenze zwischen den beiden Teilen unterhalb des gesuchten Wertes k. Der Zerlegungsprozess wird mit den Elementen a[i] ... a[r] wiederholt (vgl. Fig. 2.9).

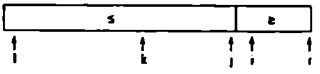


Fig. 2.10

2. Der gewählte Wert x war zu gross. Die Zerlegung muss auf dem Teil a[1] ... a[j] wiederholt werden (vgl. Abb 2.10).

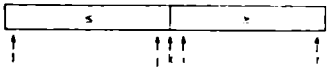


Fig. 2.11

3. j < k < i : Das Element a[k] zerlegt den Array in zwei Teile im gewünschten Verhältnis (vgl. Fig. 2.11).

Der Zerlegungsprozess ist zu wiederholen, bis der Fall 3 eintritt. Das folgende Programmstück beschreibt die Wiederholung:

```
l := 1; r := n;                                          (2.18)
while l < r do
begin x := a[k];
      "zerlege(a[l]...a[r])";
      if j < k then l := i;
      if k < i then r := j
end
```

Für den formalen Beweis der Korrektheit dieses Algorithmus wird der Leser auf den ursprünglichen Artikel von Hoare [11] verwiesen. Daraus lässt sich das ganze Programm find leicht ableiten.

```
procedure find;
   var l,r,i,j,w,x: integer;
begin l := 1; r := n;
   while l < r do
   begin x := a[k]; i := l; j := r;
      repeat {aufteilen}
         while a[i] < x do i := i+1;
         while x < a[j] do j := j-1;
         if i <= j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
              i := i+1; j := j-1
            end
      until i > j;
```

```
        if j < k then l := i;
        if k < i then r := j
    end
end {find}
```

Programm 2.12

Wenn wir davon ausgehen, dass im Mittel jede Zerlegung den Bereich halbiert, in dem die gewünschte Grösse liegt, so ist die Zahl der notwendigen Vergleiche

$$n + n/2 + n/4 + \ldots + 1 = 2n \qquad (2.19)$$

d.h. sie ist von der Grössenordnung n. Dies erklärt die Fähigkeit des Programms find zur Bestimmung des mittleren oder k-kleinsten Elementes sowie seine Ueberlegenheit über die direkten Methoden der Sortierung der ganzen Menge vor der Auswahl des k-ten Kandidaten (wobei das beste Verfahren von der Grössenordnung n*log(n) ist). Im schlimmsten Fall jedoch vermindert jeder Zerlegungsschritt die Menge der Kandidaten nur um einen und erfordert somit O(n**2) Vergleiche. Wiederum bietet die Verwendung dieses Algorithmus keinen Vorteil, wenn die Zahl der Elemente klein ist.

2.2.8. Ein Vergleich der Sortiermethoden mit Arrays

Zum Abschluss dieser Parade der Sortiermethoden wollen wir versuchen, ihre Qualitäten zu vergleichen. Wenn n die Zahl der zu ordnenden Elemente bezeichnet, sollen C und M wiederum für die Zahl der erforderlichen Vergleiche der Schlüssel, bzw. Bewegungen von Elementen stehen. Für alle drei direkten Sortiermethoden können geschlossene analytische Formeln angegeben werden. Sie sind in Tabelle 2.8 aufgeführt. Die Kolonnenüberschriften min, mit und max stehen jeweils für Minima, Maxima und die über alle n! Permutationen von n Elementen gemittelten Werte.

		min	mit	max
direktes Einfügen	C	n-1	(n**2+n-2)/2	(n**2-n)/2 -1
	M	3(n-1)	(n**2-9n-10)/4	(n**2+3n-4)/2
direkte Auswahl	C	(n**2-n)/2	(n**2-n)/2	(n**2-n)/2
	M	3(n-1)	n(ln n+0.57)	n**2/4 + 3(n-1)
direkter Austausch (Bubblesort)	C	(n**2)/2	(n**2-n)/2	(n**2-n)/2
	M	0	(n**2-n)*0.75	(n**2-n)*1.5

Tabelle 2.8

Für die höheren Methoden gibt es keine einfachen analytischen Formeln. Wesentlich ist, dass der erforderliche Rechenaufwand c[1]*n**1.2 für Shellsort und c[i]*n*log(n) für Heapsort und Quicksort ist.

Diese Formeln geben ein grobes Mass der Leistung als Funktion von n und erlauben die Klassifizierung von Sortier-Algorithmen als einfache, direkte Methoden (n**2) und höhrere oder "logarithmische" Methoden (n*log n). Für praktische Zwecke ist es jedoch nützlich, einige experimentelle Werte zur Verfügung zu haben, die einen Eindruck über die Koeffizienten c[i] vermitteln, die die verschiedenen Methoden weiter unterscheiden. Ausserdem berücksichtigen diese Formeln den rechnerischen Aufwand für andere Operationen als Vergleiche der Schlüssel und Bewegung der Elemente, wie z.B. Schleifenkontrolle, nicht. Diese Faktoren hängen sicher zu einem gewissen Grad von den einzelnen Systemen ab, aber ein Beispiel experimentell ermittelter Daten ist dennoch informativ. Tabelle 2.9 enthält die Zeiten (in Millisekunden), die die bisher besprochenen Sortiermethoden bei der Ausführung im PASCAL-System auf einer Rechenanlage CDC 6400 benötigen. Die drei Kolonnen enthalten die Zeiten für die Sortierung eines bereits geordneten Array, einer willkürlichen Permutation und eines umgekehrt geordneten Array. Die linke Zahl in jeder Kolonne gilt für 256 Elemente, die rechte für 512.

	geordnet		zufällig		umgekehrt geordnet	
direktes Einfügen	12	23	366	1444	704	2836
binäres Einfügen	56	125	373	1327	662	2490
direkte Auswahl	489	1907	509	1956	695	2675
Bubblesort	540	2165	1026	4054	1492	5931
Bubblesort mit Marke	5	8	1104	4270	1645	6542
Shakersort	5	9	961	3642	1619	6520
Shellsort	58	116	127	349	157	492
Heapsort	116	253	110	241	104	226
Quicksort	31	69	60	146	37	79
Mergesort (s. 2.3.1)	99	234	102	242	99	232

Tabelle 2.9

Die Werte unterscheiden deutlich die n**2 Methoden von den n*log n Methoden. Bemerkenswert sind folgende Punkte:

1. Die Verbesserung des binären Einfügens gegenüber dem direkten Einfügen ist wirklich unbedeutend und im Fall bereits vorhandener Ordnung sogar negativ.

2. Bubblesort ist tatsächlich die schlechteste Sortiermethode von allen, die wir verglichen haben. Die verbesserte Version "Shakersort" ist immer noch schlechter als direktes Einfügen und direkte Auswahl (mit Ausnahme des pathologischen Falls eines bereits sortierten Array).

3. Quicksort schlägt Heapsort um einen Faktor 2 bis 3. Er sortiert einen umgekehrt geordneten Array mit praktisch der gleichen Geschwindigkeit wie den bereits sortierten.

Ergänzend ist zu sagen, dass die Daten beim Sortieren von Elementen gesammelt wurden, die nur aus einem Schlüssel ohne zusätzliche Daten bestanden. Dies ist keine sehr realistische Annahme; Tabelle 2.10 zeigt den Einfluss durch Erweiterung der Grösse der Elemente. Im gewählten Beispiel belegen die zusätzlichen Daten den siebenfachen Platz des Schlüssels. Die linke Zahl in jeder Kolonne zeigt

die Zeit für das Sortieren von n = 256 Records ohne zusätzliche Daten; der rechte Wert bezieht sich auf Sortieren mit zusätzlichen Daten.

	geordnet		zufällig		umgekehrt geordnet	
direktes Einfügen	12	46	366	1129	704	2150
binäres Einfügen	56	76	373	1105	662	2070
direkte Auswahl	489	547	509	607	695	1430
Bubblesort	540	610	1026	3212	1492	5599
Bubblesort mit Marke	5	5	1104	3237	1645	5762
Shakersort	5	5	961	3071	1619	5757
Shellsort	58	186	127	373	157	435
Heapsort	116	264	110	246	104	227
Quicksort	31	55	60	137	37	75
Mergesort (s. 2.3.1)	99	196	102	195	99	187

Tabelle 2.10

Folgende Einzelheiten sind zu beachten:

1. Direkte Auswahl hat sich wesentlich verbessert und ist nun die beste der direkten Methoden.

2. Bubblesort ist immer noch mit Abstand die schlechteste Methode (sie hat sogar Boden verloren!), nur ihre "Verbesserung", Shakersort, ist im Fall des umgekehrt geordneten Array etwas schlechter!

3. Quicksort hat seine Position als schnellste Methode ausgebaut und erscheint als die bei weitem beste Sortiermethode für Arrays.

2.3. Sortieren sequentieller Files

2.3.1. Direktes Mischen

Die im vorangehenden Abschnitt behandelten Sortier-Algorithmen sind leider nicht anwendbar, wenn die zu sortierende Menge der Daten nicht in den Hauptspeicher einer Rechenanlage passt, sondern z.B. in einem peripheren und sequentiellen Speichermedium, wie einem Band, enthalten ist. In diesem Fall beschreiben wir die Daten als (sequentielles) File mit der Eigenschaft, dass zu jeder Zeit genau eine Komponente direkt zugreifbar ist. Im Vergleich zu den durch die Array-Struktur gebotenen Möglichkeiten ist dies eine starke Einschränkung, und es sind daher andere Techniken des Sortierens zu verwenden. Die bedeutendste ist Sortieren durch Mischen (merge sort). Mischen bedeutet das Zusammenfassen von zwei (oder mehreren) geordneten Sequenzen zu einer einzigen geordneten Sequenz durch fortgesetzte Auswahl aus den gerade zugreifbaren Komponenten. Mischen ist eine viel einfachere Operation als Sortieren und wird als Hilfsoperation im komplexeren Prozess des sequentiellen Sortierens verwendet. Eine Art des Sortierens durch Mischen heisst direktes Mischen und wird wie folgt beschrieben:

1. Zerlege die Sequenz a in zwei Hälften b und c.

2. Mische b und c durch Kombination einzelner Elemente zu geordneten Paaren.

3. Bezeichne die gemischte Sequenz mit a, und wiederhole Schritte 1 und 2, wobei dieses Mal die geordneten Paare zu geordneten Quadrupeln zusammenzufassen sind.

4. Wiederhole die vorangehenden Schritte durch Mischen der Quadrupel zu Octupel, und fahre damit so lange fort, indem jedesmal die Längen der gemischten Sequenzen verdoppelt werden, bis die ganze Sequenz geordnet ist.

Man betrachte als Beispiel die Sequenz

44 55 12 42 94 18 06 67 .

Beim ersten Schritt liefert die Zerlegung folgende Sequenzen

44 55 12 42
94 18 06 67 .

Das Mischen der einzelnen Komponenten (die geordnete Sequenzen der Länge 1 sind) zu geordneten Paaren ergibt

44 94 ' 18 55 ' 06 12 ' 42 67 .

Erneute Zerlegung in der Mitte und Mischen der geordneten Paare ergibt

06 12 44 94 ' 18 42 55 67 .

Eine dritte Zerlegungs- und Mischoperation führt schliesslich zum gewünschten Resultat

06 12 18 42 44 55 67 94 .

Jede Operation, die die ganze Menge der Daten einmal behandelt, heisst eine Phase, und der kleinste Teilprozess, dessen Wiederholung den Sortierprozess ergibt, heisst ein Durchlauf oder Arbeitsgang. Im obigen Beispiel umfasste das Sortieren drei Durchläufe, wobei jeder Durchlauf aus einer Zerlegungs- und einer Mischphase bestand. Zum Sortieren werden drei Bänder benötigt; der Prozess heisst daher 3-Band-Mischen.

Die Zerlegungsphasen tragen eigentlich nicht zum Sortieren bei, da sie die Reihenfolge der Elemente nicht vertauschen; in gewissem Sinn sind sie unproduktiv, obwohl sie die Hälfte aller Kopieroperationen ausmachen. Sie können durch Zusammenfassen der Zerlegungs- und Mischphasen ganz ausgeschaltet werden. Anstatt in eine einzige Sequenz zu mischen, wird die Ausgabe des Mischprozesses sofort wieder auf zwei Bänder verteilt, die dann die Quellen für den nächsten Durchlauf bilden. Im Gegensatz zum vorhergehenden 2-Phasen-Misch-Sortieren heisst diese Methode Ein-Phasen-Mischen oder ausgeglichenes Mischen. Es ist eindeutig überlegen, da nur halb so viele Kopieroperationen nötig sind; der Preis dafür ist ein viertes Band.

Wir werden ein Mischprogramm in allen Einzelheiten ausarbeiten und gehen zu Beginn davon aus, dass die Daten in einem Array dargestellt sind, der aber strenq sequentiell durchlaufen wird. Eine spätere Version einer Mischsortierung wird dann die File-Struktur verwenden und den Vergleich zweier Programme ermöglichen, der die starke Abhängigkeit der Form eines Programms von der zugrunde liegenden Datendarstellung zeigt.

Ein einzelner Array kann leicht anstelle von zwei Files verwendet werden, wenn er als zweiseitige Sequenz betrachtet wird. Anstatt von zwei Quellen-Files zu mischen, nehmen wir die Elemente von den beiden Enden des Array. Die allgemeine Form der kombinierten Misch-Aufteilungsphase lässt sich damit wie in Fig. 2.12 darstellen.

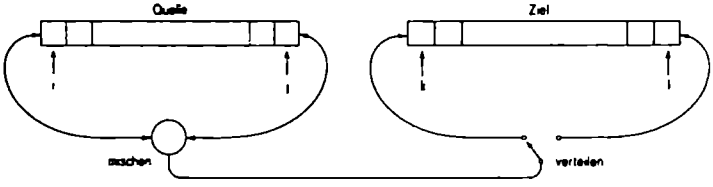


Fig. 2.12: Array Sortieren durch Mischen

Das Ziel der gemischten Elemente wird nach jedem geordneten Paar im ersten Durchlauf, nach jedem geordneten Quadrupel im zweiten Durchlauf, usw., geändert, so dass die beiden Ziel-Sequenzen, dargestellt durch die beiden Enden eines einzelnen Array, gleichmässig aufgefüllt werden. Nach jedem Durchlauf vertauschen die beiden Arrays ihre Rollen, die Quelle wird zum Ziel und umgekehrt.

Das Programm lässt sich nun weiter vereinfachen, indem man die beiden begrifflich verschiedenen Arrays zu einem einzigen Array doppelter Grösse zusammenfasst. Somit werden die Daten dargestellt durch

```
a : array [1..2*n] of item                    (2.20)
```

und die Indizes i und j bezeichnen die beiden Quellenelemente, während k und l die beiden Ziele angeben (siehe Fig. 2.12). Die

Ausgangswerte sind natürlich die Werte a[1]...a[n]. Eine Boolesche Variable up wird benötigt, um die Richtung des Datenflusses anzuzeigen; "up = true" bedeutet, dass im gegenwärtigen Durchlauf die Komponenten a[1] ... a[n] zu den Variablen a[n+1] ... a[2n] "hinauf" (up) bewegt werden, während "up = false" den Transport von a[n+1]...a[2n] nach a[1]...a[n] "hinunter" anzeigt. Der Wert von up wechselt immer zwischen aufeinanderfolgenden Durchläufen. Schliesslich wird eine Variable p zur Angabe der Länge der zu mischenden Teil-Sequenz eingeführt. Ihr Wert ist zu Beginn 1 und wird für jeden folgenden Durchlauf verdoppelt. Zur Vereinfachung nehmen wir an, dass n immer eine Potenz von 2 ist. Damit hat die erste Version eines direkten Mischprogramms folgende Form:

```
procedure mergesort;                                        (2.21)
   var i,j,k,l: index;
       up: Boolean; p: integer;
begin up := true; p := 1;
   repeat {initialisiere die Indizes}
      if up then
         begin i := 1; j := n; k := n+1; l := 2*n
         end else
         begin k := 1; l := n; i := n+1; j := 2*n
         end;
      "mische die p-Tupel der i- und j-Sequenzen zu
       k- und l-Sequenzen";
      up := not up; p := 2*p
   until p = n
end
```

Im nächsten Entwicklungsschritt verfeinern wir die (in Anführungszeichen stehende und) in natürlicher Sprache ausgedrückte Anweisung. Offenbar ist dieser Mischdurchlauf mit n Elementen selbst eine Sequenz von Mischoperationen von Teil-Sequenzen, d.h. von p-Tupeln. Zwischen jedem solchen teilweisen Mischvorgang wird der Zielpunkt vom unteren zum oberen Ende des Ziel-Array umgeschaltet oder umgekehrt, um gleichmässige Verteilung auf die beiden Ziel-Sequenzen zu garantieren. Kommen die gemischten Elemente in das untere Ende des Ziel-Array, so ist der Ziel-Index k, und k wird nach jeder Bewegung eines Elementes um 1 erhöht. Werden sie an das obere Ende des Ziel-Array gebracht, so ist l der Ziel-Index, und l

ist dann nach jeder Bewegung um 1 zu verkleinern. Um die gegenwärtige Mischanweisung zu vereinfachen, legen wir fest, dass das Ziel immer durch k bezeichnet werden soll, indem wir die Werte der Variablen k und l nach jedem Durchlauf vertauschen. Wir bezeichnen die immer zu verwendende Schrittweite durch h, wobei h entweder 1 oder -1 ist. Dies führt zu folgender Verfeinerung.

```
h := 1; m := n;                                              (2.22)
{m = Anzahl der zu mischenden Elemente}
repeat q := p; r := p; m := m-2*p;
   "mische q Elemente von i mit r Elementen von j ,
    Ziel-Index ist k mit Schrittweite h";
   h := -h;
   "vertausche k und l"
until m = 0
```

Im nächsten Verfeinerungsschritt wird die eigentliche Mischanweisung formuliert. Dabei haben wir zu berücksichtigen, dass der Rest der einen nach dem Mischen nicht leeren Teil-Sequenz an die Ausgabe-Sequenz durch eine Folge einfacher Kopieroperationen anzuhängen ist.

```
while (q>0) and (r>0) do                                     (2.23)
begin  {wähle ein Element von i oder j}
   if a[i].key < a[j].key then
      begin "bringe ein Element von i nach k;
             schiebe i und k weiter";  q := q-1
      end else
      begin "bringe ein Element von j nach k;
             schiebe j und k weiter";  r := r-1
      end
end;
"kopiere Rest der i-Sequenz";
"kopiere Rest der j-Sequenz"
```

Nach der weiteren Verfeinerung der Kopieroperationen des Restes ist das Programm in allen Einzelheiten dargelegt. Bevor wir es vollständig ausschreiben, wollen wir die Einschränkung aufheben, wonach n eine Potenz von 2 sein muss. Welche Teile des Algorithmus werden durch die Abschwächung dieser Einschränkung betroffen? Man

überzeugt sich leicht, dass die beste Art der Behandlung einer allgemeineren Situation möglichst langes Festhalten an der alten Methode ist. In unserem Beispiel bedeutet dies, dass wir fortfahren, p-Tupel zu mischen, bis die Reste der Quellen-Sequenzen in der Länge kleiner als p sind. Der einzige beeinflusste Teil sind daher die Anweisungen, die die Werte von q und r, die Längen der zu mischenden Sequenzen bestimmen. Die nachstehenden vier Anweisungen ersetzen die drei Anweisungen

```
q := p; r := p; m := m-2*p
```

und, wie der Leser sich selbst überzeugen kann, stellen wirklich eine Implementation der oben festgelegten Strategie dar; man beachte, dass m die gesamte Zahl der restlichen von beiden Quellen-Sequenzen zu mischenden Elemente angibt:

```
if m >= p then q := p else q := m; m := m-q;
if m >= p then r := p else r := m; m := m-r
```

Um zusätzlich Termination des Programms zu garantieren, muss die Bedingung p = n, die die "äussere" Schl ife kontrolliert, geändert werden zu p >= n. Nach diesen Aenderungen können wir nun den ganzen Algorithmus als vollständiges Programm 2.13 angeben.

```
procedure mergesort;
   var i,j,k,l,t: index;
      h,m,p,q,r: integer;  up: Boolean;
   {beachte, dass a 2*n Elemente besitzt}
begin up := true; p := 1;
   repeat h := 1; m := n;
      if up then
      begin i := 1; j := n; k := n+1; l := 2*n
      end else
      begin k := 1; l := n; i := n+1; j := 2*n
      end ;
      repeat  {mische einen Lauf von i und j auf k}
         {q und r: Länge des i-ten bzw. j-ten Laufs}
         if m >= p then q := p else q := m;   m := m-q;
         if m >= p then r := p else r := m;   m := m-r;
         while (q<>0) and (r<>0) do
```

```
        begin {mische}
           if a[i].key < a[j].key then
           begin a[k] := a[i]; k := k+h; i := i+1; q := q-1
           end else
           begin a[k] := a[j]; k := k+h; j := j-1; r := r-1
           end
        end ;
        if q = 0 then {kopiere Rest des j-ten Laufs}
           while r > 0 do
           begin a[k] := a[j]; k := k+h; j := j-1; r := r-1
           end
        else {r = 0, kopiere Rest des i-ten Laufs}
           while q > 0 do
           begin a[k] := a[i]; k := k+h; i := i+1; q := q-1
           end ;
        h := -h; t := k; k := l; l := t
     until m = 0;
     up := not up; p := 2*p
  until p >= n;
  if not up then
     for i := 1 to n do a[i] := a[i+n]
end {mergesort}
```

Programm 2.13: Direktes Mischsortieren am Array

Analyse des Mischsortierens: Da jeder Durchlauf p verdoppelt und da jede Sortierung abgeschlossen ist, sobald p >= n, umfasst das Mischsortieren ganz(log n) Durchläufe. Jeder Durchlauf kopiert nach Definition die ganze Menge der n Elemente genau einmal. Folglich ist die gesamte Zahl aller Bewegungen genau

$$M = n * \text{ganz}(\log n) \; . \qquad (2.24)$$

Die Zahl C der Schlüsselvergleiche ist sogar kleiner als n, da die Kopieroperationen für den Rest keine Vergleiche erfordern. Da jedoch die Technik des Mischsortierens gewöhnlich zusammen mit peripheren Speichermedien verwendet wird, übertrifft der Aufwand der Bewegungsoperationen den Aufwand der Vergleiche oft um mehrere Grössenordnungen. Die detaillierte Analyse der Zahl der Vergleiche ist daher kaum von praktischer Bedeutung.

Der Algorithmus des Mischsortierens schneidet offensichtlich gut ab, sogar im Vergleich mit den im vorhergehenden Abschnitt erörterten höheren Techniken des Sortierens. Aber der verwaltungsmässige Aufwand für die Behandlung der Indizes ist relativ gross, und der entscheidende Nachteil ist der Speicheraufwand für 2*n Elemente. Aus diesem Grund wird Sortieren durch Mischen selten auf Arrays angewendet, d.h. auf Daten, die im Hauptspeicher sind. Werte zum Vergleich des zeitlichen Verhaltens dieses Algorithmus sind in den letzten Zeilen der Tabellen 2.9 und 2.10 aufgeführt. Sie schneiden gegenüber Heapsort günstig, gegenüber Quicksort ungünstig ab.

2.3.2. Natürliches Mischen

Beim direkten Mischen wird aus einer anfangs vorhandenen, teilweisen Ordnung kein Nutzen gezogen. Die Länge aller gemischten Teil-Sequenzen im k-ten Durchlauf ist (kleiner oder) gleich 2**k, unabhängig davon, ob bereits längere Teil-Sequenzen geordnet sind und gemischt werden könnten. Tatsächlich könnten irgend zwei geordnete Teil-Sequenzen mit Längen m und n direkt in eine Sequenz mit m+n Elementen gemischt werden. Ein Mischsortieren, das immer die beiden längsten möglichen Teil-Sequenzen mischt, heisst *natürliches Mischsortieren*.

Eine geordnete Teil-Sequenz wird oft *String* genannt. Da das Wort String jedoch häufiger zur Beschreibung von Zeichenfolgen gebraucht wird, schliessen wir uns in unserer Terminologie Knuth an und verwenden das Wort *Lauf* (run) statt *String*, wenn wir uns auf geordnete Teil-Sequenzen beziehen. Wir nennen eine Teil-Sequenz a[i]...a[j] einen *maximalen Lauf* oder kurz einen Lauf, wenn sie folgende Bedingungen erfüllt:

1. a[k] <= a[k+1] für k = i...j-1 (2.25)
2. a[i-1] > a[i] und
3. a[j] > a[j+1]

Eine natürliche Mischsortierung mischt also (maximale) Läufe statt feste Sequenzen mit vorbestimmter Länge. Läufe haben die Eigenschaft, dass beim Mischen von zwei Sequenzen mit n Läufen, eine

einzige Sequenz mit genau n Läufen entsteht. Somit wird die gesamte Zahl der Läufe in jedem Durchlauf halbiert, und die Zahl der benötigten Bewegungen von Elementen beträgt im schlimmsten Fall n*ganz(log(n)), ist aber im Durchschnitt eher kleiner. Die zu erwartende Zahl von Vergleichen ist aber viel grösser, da zusätzlich zu den für die Auswahl der Elemente nötigen Vergleichen weitere Vergleiche zwischen aufeinanderfolgenden Elementen jedes Files zur Bestimmung des Endes jedes Laufs erforderlich sind.

Unsere nächste Programmierübung entwickelt einen Algorithmus für natürliches Mischen auf die gleiche schrittweise Art, die zur Erklärung des Algorithmus des direkten Mischens verwendet wurde. Sie benutzt die Struktur des sequentiellen Files anstelle des Array und stellt eine nicht ausgeglichene 2-Phasen, 3-Band-Mischsortierung dar. Wir nehmen an, dass die ursprüngliche Sequenz der Elemente als File c gegeben ist, auf dem auch die geordnete Ausgabe erscheinen wird. (Natürlich werden in der Praxis der Datenverarbeitung aus Gründen der Sicherheit die ursprünglichen Werte zuerst vom Originalband auf ein File c kopiert.) Die beiden Hilfsbänder sind a und b. Jeder Durchlauf umfasst eine Verteilungsphase (distribution phase), die Läufe gleichmässig von c auf a und b verteilt, und eine Mischphase (merge phase), die Läufe von a und b auf c mischt. Dieser Prozess wird in Fig. 2.13 illustriert.

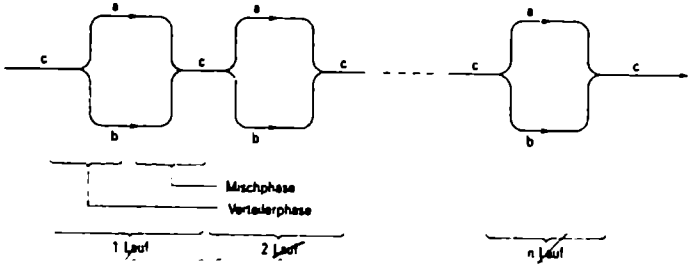


Fig. 2.13: Verteilungs- und Mischphasen beim Sortieren durch Mischen

Als Beispiel zeigt Tabelle 2.11 das File c im ursprünglichen Zustand (Zeile 1) und dann nach jedem Durchlauf (Zeilen 2 bis 4) in einer natürlichen Mischsortierung mit 20 Zahlen. Man beachte, dass nur drei Durchläufe benötigt werden.

```
17 31'05 59'13 41 43 67'11 23 29 47'03 07 71'02 19 57'37 61
05 17 31 59'11 13 23 29 41 43 47 67'02 03 07 19 57 71'37 61
05 11 13 17 23 29 31 41 43 47 59 67'02 03 07 19 37 57 61 71
02 03 05 07 11 13 17 19 23 29 31 37 41 43 47 57 59 61 67 71
```

Tabelle 2.11

Das Sortieren ist beendet, sobald auf c nur noch ein Lauf ist. (Wir nehmen an, dass mindestens ein nicht leerer Lauf auf dem ursprünglichen File vorhanden war.) Wir verwenden daher eine Variable l, um die Zahl der auf das File c gemischten Läufe zu zählen. Definieren wir die globalen Einheiten als

```
type tape = file of item;                                    (2.26)
var c: tape
```

so kann das Programm folgendermassen formuliert werden:

```
procedure naturalmerge;                                      (2.27)
   var l: integer;
       a,b: tape;
begin
   repeat rewrite(a); rewrite(b); reset(c);
          distribute;
          reset(a); reset(b); rewrite(c);
          l := 0; merge
   until l = 1
end
```

Die beiden Phasen ergeben sich als völlig getrennte Anweisungen. Sie sind nun zu verfeinern, d.h. in weitere Einzelheiten aufzugliedern. Die verfeinerten Beschreibungen können entweder direkt eingesetzt oder als Prozeduren beschrieben werden, und die abgekürzten Anweisungen sind dann als Prozedur-Aufrufe zu verstehen. Wir wählen diesmal das letztere und definieren

```
procedure distribute; {von c nach a und b}                   (2.28)
begin
   repeat copyrun(c,a);
      if not eof(c) then copyrun(c,b)
```

```
      until eof(c)
   end
und
   procedure merge;                                          (2.29)
   begin {von a und b nach c}
      repeat mergerun; l := l+1
      until eof(b);
      if not eof(a) then
         begin copyrun(a,c); l := l+1
         end
   end
```

Diese Art der Verteilung bringt entweder eine gleiche Zahl von Läufen auf die Files a und b oder auf das File a einen Lauf mehr als auf das File b. Da entsprechende Paare von Läufen gemischt werden, kann ein Lauf auf dem File a übrigbleiben, der dann einfach zu kopieren ist. Die Prozeduren merge und distribute sind unter Verwendung der untergeordneten Prozeduren mergerun und copyrun formuliert, deren Aufgaben aus ihrem Namen hervorgehen: Mischen bzw. Kopieren eines Laufs. Diese Prozeduren werden nun in weiteren Einzelheiten ausgearbeitet; sie bedingen die Einführung einer globalen Booleschen Variablen eor, die angibt, ob das Ende eines Laufs (end of run) erreicht ist oder nicht.

```
   procedure copyrun(var x,y: tape);                         (2.30)
   begin  {kopiere einen Lauf von x nach y}
      repeat copy(x,y) until eor
   end

   procedure mergerun;                                       (2.31)
   begin  {mische einen Lauf von a und b nach c}
      repeat  if a↑.key < b↑.key then
                 begin copy(a,c);
                    if eor then copyrun(b,c)
                 end else
                 begin copy(b,c);
                    if eor then copyrun(a,c)
                 end
      until eor
   end
```

Der Prozess des Vergleichens und Auswählens von Schlüsseln beim Mischen eines Laufs ist beendet, sobald einer der beiden Läufe verarbeitet ist. Danach ist der andere (noch nicht abgearbeitete) Lauf in den sich ergebenden Lauf durch Kopieren des Restes zu bringen. Dies wird durch den Aufruf der Prozedur copyrun getan.

Die beiden Prozeduren sind unter Verwendung der untergeordneten Prozedur copy definiert, die ein Element von einem Quellen-File x auf ein Ziel-File y bringt und festlegt, ob das Ende eines Laufs erreicht ist. Sie lässt sich leicht ausdrücken durch die einfachen Prozeduren read und write. Zur Bestimmung des Endes eines Laufs muss das zuletzt gelesene (kopierte) Element (oder zumindest sein Schlüssel) für den Vergleich mit seinem Nachfolger aufbewahrt werden.

```
procedure copy(var x,y: tape);                    (2.32)
   var buf: item;
begin read(x,buf); write(y,buf);
   if eof(x) then eor := true
             else eor := buf.key > x↑.key
end
```

Damit ist die Entwicklung der Prozedur zur natürlichen Mischsortierung abgeschlossen. Leider ist dieses Programm nicht korrekt, wie der sorgfältige Leser vielleicht schon gemerkt hat. Das Programm sortiert bestimmte Fälle nicht richtig. Man betrachte z.B. folgende Eingabe-Sequenz:

3 2 5 11 7 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67 .

Verteilung aufeinanderfolgender Läufe auf die Files a und b ergibt

a = 3 ' 7 13 19 ' 29 37 43 ' 57 61 71 '
b = 2 5 11 ' 17 23 31 ' 41 47 59 ' 67 .

Diese Sequenzen werden leicht zu einem einzigen Lauf gemischt, wonach das Sortieren erfolgreich beendet ist. Obwohl das Beispiel nicht zu fehlerhaftem Verhalten des Programms führt, macht es uns darauf aufmerksam, dass reines Verteilen von Läufen auf verschiedene Files zu einer kleineren Zahl von Ausgabe-Läufen führen kann, als Eingabe-Läufe vorhanden waren. Dies kann dann eintreten, wenn

das erste Element des (i+2)-ten Laufs grösser ist als das letzte Element des i-ten Laufs und somit zwei Läufe automatisch zu einem Lauf zusammengefasst werden.

Obwohl die Prozedur distribute anscheinend gleich viele Läufe auf beide Files bringt, kann also die tatsächliche Zahl der auf a und b entstehenden Läufe wesentlich verschieden sein. Unsere Prozedur merge aber mischt nur Paare von Läufen und bricht ab, sobald das File b gelesen ist, wobei der Rest des einen Files verloren geht. Man betrachte die folgenden Eingabe-Daten, die in zwei aufeinanderfolgenden Durchläufen geordnet (und abgeschnitten) werden:

```
17 19 13 57 23 29 11 59 31 37  7 61 41 43  5 67 47 71  2  3
13 17 19 23 29 31 37 41 43 47 57 71 11 59
11 13 17 19 23 29 31 37 41 43 47 57 59 71
```

Tabelle 2.12

Das Beispiel dieses Programmierfehlers ist typisch für viele Situationen beim Programmieren. Der Fehler wird verursacht durch das Uebersehen einer möglichen Konsequenz einer anscheinend einfachen Operation. Er ist auch in dem Sinn typisch, als es verschiedene Wege zur Behebung des Fehlers gibt und man einen davon wählen muss. Oft gibt es zwei wesentlich und grundlegend verschiedene Möglichkeiten:

1. Wir erkennen, dass die Operation der Verteilung nicht korrekt programmiert wurde und dass sie die Zahl der Läufe nicht gleichmässig (oder höchstens mit dem Unterschied 1) verteilt. Wir halten uns an die ursprüngliche Aufgabe der Operation und verbessern dementsprechend die falsche Prozedur.

2. Wir erkennen, dass die Verbesserung des falschen Teils weitreichende Aenderungen verursacht, und versuchen, ob andere Teile des Algorithmus geändert werden können, so dass sie zu dem momentan falschen Teil passen.

Im allgemeinen scheint der erste Weg der sicherere, sauberere und ehrlichere Weg zu sein, der uns bis zu einem gewissen Grad vor späteren Folgen von Uebersehenen, verworrenen Nebenwirkungen

schützt. Im allgemeinen (und zurecht) ist deshalb dieser Weg zur Lösung zu empfehlen.

Es ist jedoch darauf hinzuweisen, dass manchmal auch die zweite Möglichkeit nicht völlig ausser acht zu lassen ist. Aus diesem Grund arbeiten wir dieses Beispiel weiter aus und zeigen eine Korrektur, indem wir die Prozedur merge anstelle der eigentlich falschen Prozedur distribute ändern.

Das bedeutet, dass wir das Schema der Verteilung unverändert lassen und die Bedingung aufheben, dass die Läufe gleichmässig verteilt seien. Dies kann eine nicht optimale Leistung zur Folge haben. Die Leistung im schlimmsten Fall bleibt aber unverändert, und ausserdem ist der Fall einer äusserst ungleichmässigen Verteilung statistisch sehr unwahrscheinlich. Effizienz-Betrachtungen sind deshalb kein ernstes Argument gegen diese Lösung.

Wenn die Bedingung der Gleichverteilung der Läufe nicht mehr existiert, muss die Prozedur merge so geändert werden, dass sie nach Erreichen des Endes eines Files den ganzen Rest des anderen Files anstatt höchstens einen Lauf kopiert.

Diese Aenderung ist direkt und sehr einfach im Vergleich zu irgendeiner Aenderung im Verteilungsschema. (Der Leser sollte sich unbedingt selbst von der Richtigkeit dieser Behauptung überzeugen.) Die geänderte Version des Misch-Algorithmus ist Teil des vollständigen Programms 2.14.

```
program mergesort(input,output);
{3-Band, 2-Phasen natürliches Mischsortieren}
type item = record key: integer
                {andere Felder hier definiert}
            end ;
     tape = file of item;
var c: tape; n: buf: item;

procedure list(var f: tape);
   var x: item;
begin reset(f);
   while not eof(f) do
```

```
      begin read(f,x); write(output, x.key: 4)
      end ;
   writeln
end {list} ;

procedure naturalmerge;
   var l: integer;   {Zahl gemischter Läufe}
       eor: Boolean; {end-of-run Anzeige}
       a,b: tape;

   procedure copy(var x,y: tape);
      var buf: item;
   begin read(x, buf); write(y,buf);
      if eof(x) then eor := true else eor := buf.key > x↑.key
   end ;

   procedure copyrun(var x,y: tape);
   begin {kopiere einen Lauf von x auf y}
      repeat copy(x,y) until eor
   end ;

   procedure distribute;
   begin {von c auf a und b}
      repeat copyrun(c,a);
         if not eof(c) then copyrun(c,b)
      until eof(c)
   end ;

   procedure mergerun;
   begin {von a und b auf c}
      repeat
         if a↑.key < b↑.key then
         begin copy(a,c);
            if eor then copyrun(b,c)
         end else
         begin copy(b,c);
            if eor then copyrun(a,c)
         end
      until eor
   end ;
```

```
   procedure merge;
   begin {von a und b auf c}
      repeat mergerun; l := l+1
      until eof(a)  or  eof(b);
      while not eof(a) do
      begin copyrun(a,c); l := l+1
      end ;
      while not eof(b) do
      begin copyrun(b,c); l := l+1
      end ;
      list(c)
   end ;

begin
   repeat rewrite(a); rewrite(b); reset(c);
      distribute;
      reset(a); reset(b); rewrite(c);
      l := 0; merge
   until l = 1
end {naturalmerge};

begin {Hauptprogramm.
       Eingabe-Sequenz ist mit 0 abzuschliessen}
   rewrite(c); read(buf.key);
   repeat write(c, buf); read(buf.key)
   until buf.key = 0;
   list(c);
   naturalmerge;
   list(c)
end .
```

Programm 2.14: Natürliches Mischsortieren

2.3.3. Ausgeglichenes N-Weg-Mischen

Der Aufwand für sequentielles Sortieren ist proportional zur Zahl der benötigten Durchläufe, da nach Definition bei jedem Durchlauf die ganze Menge der Daten kopiert wird. Ein Weg zur Reduzierung dieser Zahl ist die Verteilung der Läufe auf mehr als zwei Files.

Das Mischen von r Läufen, die auf N Bänder gleich verteilt sind, ergibt eine Sequenz mit r/N Läufen. Ein zweiter Durchlauf vermindert ihre Zahl auf r/N**2, ein dritter Durchlauf auf r/N**3, und nach k Durchläufen bleiben r/N**k Läufe übrig. Zum Sortieren von n Elementen durch ein N-Weg-Mischen werden also insgesamt K = ganz(logN(n)) Durchläufe benötigt. Da jeder Durchlauf n Kopieroperationen erfordert, beträgt die Gesamtzahl der Kopieroperationen im schlimmsten Fall

M = n*ganz(logN n) .

Als nächste Programmierübung wollen wir ein auf N-Weg-Mischen beruhendes Programm entwickeln. Um das Programm von der vorhergehenden Prozedur für natürliches 2-Phasen-Mischen stärker abzusetzen, werden wir das N-Weg-Mischen als eine ausgeglichene 1-Phasen-Mischsortierung (Mergesort) formulieren. Dies bedingt, dass in jedem Durchlauf eine gleiche Zahl von Eingabe- und Ausgabe-Files zur Verfügung stehen, auf die aufeinanderfolgende Läufe abwechselnd verteilt werden. Verwendet man N Files, so beruht der Algorithmus auf einem N/2-Weg-Mischen unter der Voraussetzung, dass N eine gerade Zahl ist. Entsprechend der bisher verfolgten Strategie, wollen wir uns nicht um das automatische Mischen von zwei aufeinanderfolgenden, auf das gleiche Band verteilten Läufen kümmern. Folglich haben wir ein Mischprogramm zu entwerfen, das nicht unbedingt gleiche Anzahl von Läufen auf den Eingabe-Bändern erwartet.

In diesem Programm begegnen wir zum ersten Mal einer natürlichen Anwendung der Datenstruktur eines Array von Files. Es ist tatsächlich überraschend, wie sehr sich das folgende Programm wegen des Uebergangs von einem 2-Weg- zu einem N-Weg-Mischen vom vorhergehenden unterscheidet. Die Aenderung ist im wesentlichen dadurch bedingt, dass der Mischvorgang nicht mehr einfach abgeschlossen werden kann, wenn einer der beiden Eingabe-Läufe abgearbeitet ist. Statt dessen ist eine Liste immer noch aktiver, d.h. noch nicht abgearbeiteter Läufe zu führen. Eine weitere Komplikation entsteht durch das notwendige Umschalten zwischen den Gruppen der Ein- und Ausgabe-Bänder nach jedem Durchlauf.

Wir beginnen mit der Definition eines Typs, zusätzlich zu den vertrauten Typen item und tape.

```
tapeno = 1..n                                          (2.33)
```

Bandnummern werden offensichtlich zur Indizierung des Array von Files von Elementen verwendet. Die ursprüngliche Sequenz von Elementen sei gegeben durch eine Variable

```
f0 : tape ,                                            (2.34)
```

und es stehen N Bänder mit geradzahligem N für den Sortierprozess zur Verfügung:

```
f: array [tapeno] of tape                              (2.35)
```

Eine empfehlenswerte Technik zur Behandlung des Problems des Umschaltens zwischen den Bändern ist die Einführung einer Abbildung für die Band-Indizes. Anstatt direkt durch seinen Index i, wird ein Band über die Abbildung t adressiert, d.h. für jedes

```
f[i]      schreiben wir      f[t[i]] ,
```

wobei die Abbildung definiert ist als

```
t: array [tapeno] of tapeno .                          (2.36)
```

Gilt ursprünglich t[i] = i für alle i, so besteht ein Umschalten lediglich im Austauschen der Paare der Abbildungs-Komponenten

```
t[1]  <=>  t[nh+1]
t[2]  <=>  t[nh+2]
      ...
t[nh] <=>  t[n] ,
```

wobei nh = n/2 . Folglich können wir immer

```
f[t[1]], ... , f[t[nh]]
```

als Eingabe-Bänder und

```
f[t[nh+1]], ..., f[t[n]]
```

als Ausgabe-Bänder betrachten. (Im folgenden wird f[t[j]] in Kommentaren einfach "Band j" genannt.) Der Algorithmus kann nun

zunächst folgendermassen formuliert werden:

```
procedure tapemergesort;                                    (2.37)
   var i,j: tapeno;
      l: integer; {Anzahl der verteilten Läufe}
      t: array [tapeno] of tapeno;
begin {verteile ursprüngliche Läufe auf t[1]...t[nh]}
   j := nh; l := 0;
   repeat if j < nh then j := j+1 else j := 1;
      "kopiere einen Lauf von f0 auf Band j";
      l := l+1
   until eof(f0);
   for i := 1 to n do t[i] := i;
   repeat {mische von t[1]...t[nh] auf t[nh+1]...t[n]}
      "spule die Eingabe-Bänder zurück";
      l := 0;
      j := nh+1; {j = Index des Ausgabe-Bandes}
      repeat l := l+1;
         "mische einen Lauf von den Eingaben nach t[j]"
         if j < n then j := j+1 else j := nh+1
      until "alle Eingaben sind abgearbeitet";
      "schalte die Bänder um"
   until l = 1;
   {Resultat auf Band 1}
end
```

Zuerst verfeinern wir die in der ursprünglichen Verteilung der Läufe verwendetete Kopieroperation; wir führen wiederum eine Hilfsvariable zur Speicherung des zuletzt gelesenen Elementes ein:

```
buf: item
```

und ersetzen "kopiere einen Lauf von f0 auf Band j" durch die Anweisung

```
repeat read(f0,buf);                                        (2.38)
       write(f[j],buf)
until (buf.key > f0↑.key) or eof(f0)
```

Das Kopieren eines Laufs ist beendet, wenn entweder das Element des nächsten Laufs gefunden wird (buf.key > f0↑.key) oder das Ende des ganzen Eingabe-Files erreicht ist (eof(f0)).

Im gegenwärtigen Sortier-Algorithmus sind noch die Anweisungen

(1) "Spule die Eingabe-Bänder zurück"
(2) "Mische einen Lauf von den Eingaben auf t[j]"
(3) "Schalte die Bänder um"

und das Prädikat

(4) "alle Eingaben sind abgearbeitet"

in weiteren Einzelheiten auszuführen. Zunächst müssen wir die gegenwärtigen Eingabe-Files richtig bestimmen. Insbesondere kann die Zahl der "aktiven" Eingabe-Files kleiner als n/2 sein. Tatsächlich gibt es höchstens so viele Quellen wie Läufe; das Sortieren ist abgeschlossen, sobald nur ein File übrigbleibt. Damit ist es möglich, dass bei der Initialisierung des letzten Sortierdurchgangs weniger als nh Läufe vorhanden sind. Wir führen daher eine Variable, etwa k1, ein zur Bezeichnung der tatsächlich verwendeten Eingabe-Files. Wir schliessen die Initialisierung von k1 folgendermassen in die Anweisung "spule die Eingabe-Bänder zurück" ein:

```
if 1 < nh then k1 := 1 else k1 := nh;
for i := 1 to k1 do reset(f[t[i]])
```

Natürlich muss Anweisung (2) k1 jedesmal verkleinern, sobald eine Eingabe-Quelle erschöpft ist. Prädikat (4) kann nun einfach ausgedrückt werden durch die Relation k1 = 0 .

Anweisung (2) ist schwieriger auszuarbeiten: sie besteht aus wiederholter Auswahl des kleinsten Schlüssels aus den verfügbaren Quellen und dem anschliessenden Transport des entsprechenden Elementes an das Ziel, d.h. auf das gegenwärtige Ausgabe-Band. Dieser Vorgang wird wiederum komplizierter durch das notwendige Bestimmen des Endes eines jeden Laufs. Das Ende eines Laufs ist erreicht, wenn entweder (a) der nachfolgende Schlüssel kleiner ist als der gegenwärtige oder (b) das Ende des Quellen-Files erreicht ist. Im letzteren Fall wird das Band durch Verkleinern von k1 ausgeschlossen, im ersteren Fall wird der Lauf durch Ausschliessen des Files von der weiteren Auswahl der Elemente abgeschlossen, aber nur bis das Erstellen des gegenwärtigen Ausgabe-Laufs beendet ist.

Daraus ergibt sich die Notwendigkeit einer zweiten Variablen, sagen wir k2, zur Bezeichnung der Zahl der gegenwärtig für die Auswahl des nächsten Elementes zur Verfügung stehenden Bänder. Dieser Wert wird zu Anfang gleich k1 gesetzt und jedesmal verkleinert, wenn ein Lauf nach Bedingung (a) beendet ist.

Leider ist das Einführen von k2 nicht ausreichend: die Kenntnis der Zahl der Bänder genügt nicht. Wir müssen auch genau wissen, welche Bänder noch im Spiel sind. Eine naheliegende Lösung ist die Verwendung eines Array mit Booleschen Komponenten, die die Verfügbarkeit der Bänder angeben. Wir wählen aber eine andere Methode, die zu einer effizienteren Auswahlprozedur führt, welche ja nach allem der häufigst wiederholte Teil des ganzen Algorithmus ist. Anstelle eines Booleschen Array führen wir eine zweite Abbildung der Bänder, sagen wir ta, ein. Diese Abbildung wird anstelle von t benutzt, so dass ta[1]...ta[k2] die Indizes der verfügbaren Bänder sind. Anweisung (2) kann somit geschrieben werden als:

```
k2 := k1;                                                    (2.39)
repeat "wähle den kleinsten Schlüssel,
    ta[mx] sei seine Bandnummer";
    read(f[ta[mx]],buf);
    write(f[t[j]],buf);
    if eof(f[ta[mx]]) then "schliesse Band aus" else
    if buf.key > f[ta[mx]]↑.key then "schliesse Lauf ab"
until k2 = 0
```

Da die bei jeder Rechenanlage zur Verfügung stehende Zahl von Bandeinheiten gewöhnlich klein ist, kann der im nächsten Verfeinerungsschritt im einzelnen auszuführende Auswahl-Algorithmus auch eine direkte lineare Suche sein. Die Anweisung "schliesse Band aus" impliziert eine Verkleinerung von k1 und auch k2 sowie eine erneute Zuweisung der Indizes in der Abbildung ta, während die Anweisung "schliesse den Lauf ab" nur k2 verkleinert und die Komponenten von ta entsprechend umordnet. Die Einzelheiten sind in Programm 2.15 enthalten, das eine letzte Verfeinerung von (2.37) bis (2.39) ist. Man beachte, dass die Bänder durch die Prozedur rewrite zurückgespult werden, sobald ihr letzter Lauf gelesen wurde. Die Anweisung "schalte die Bänder um" ist entsprechend den früher gegebenen Erläuterungen ausgearbeitet.

```
program balancedmerge(output);
{Ausgeglichenes n-Weg-Misch-Sortieren}
const n = 6; nh = 3;          {Zahl der Bänder}
type item = record
               key: integer
            end ;
   tape = file of item;
   tapeno = 1..n;
var leng, rand: integer;        {Zum Erstellen des Files}
   eot: Boolean;
   buf: item;
   f0: tape;  {Eingabe-Band mit Zufallszahlen}
   f: array [1..n] of tape;

procedure list(var f: tape; n: tapeno);
   var z: integer;
begin writeln(' tape', n:2); z := 0;
   while not eof(f) do
   begin read(f, buf); write(output, buf.key: 5); z := z+1;
      if z = 25 then
         begin writeln(output); z := 0;
         end
   end ;
   if z <> 0 then writeln(output); reset(f)
end {list} ;

procedure tapemergesort;
   var i,j,mx,tx: tapeno;
      k1,k2,l: integer;
      x, min: integer;
      t, ta: array [tapeno] of tapeno;
begin {verteile ursprüngliche Läufe auf t[1] .. t[nh]}
   for i := 1 to nh do  rewrite(f[i]);
   j := nh; l := 0;
   repeat if j < nh then j := j+1 else j := 1;
      {kopiere einen Lauf von f0 auf Band j}
      l := l+1;
      repeat read(f0, buf); write(f[j], buf)
      until (buf.key > f0↑.key)  or  eof(f0)
   until eof(f0);
```

```
for i := 1 to n do t[i] := i;
repeat {mische von t[1] ... t[nh] auf t[nh+1] ... t[n]}
   if l < nh then k1 := l else k1 := nh;
   {k1 = Zahl der Eingabe-Bänder in dieser Phase}
   for i := 1 to k1 do
   begin reset(f[t[i]]); list(f[t[i]], t[i]); ta[i] := t[i]
   end ;
   l := 0;   {l = Zahl der gemischten Läufe}
   j := nh+1;    {j = Index des Ausgabe-Bandes}
   repeat {mische einen Lauf von t[1] ... t[k1] auf t[j]}
      k2 := k1; l := l+1;  {Zahl der aktiven Eingabe-Bänder}
      repeat {wähle kleinstes Element}
         i := 1; mx := 1; min := f[ta[1]]↑.key;
         while i < k2 do
         begin i := i+1; x := f[ta[i]]↑.key;
               if x < min then
               begin min := x;  mx := i
               end
         end ;
         {ta[mx] hat minimales Element, bringe es auf t[j]}
         read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
         write(f[t[j]], buf);
         if eot then
         begin rewrite(f[ta[mx]]);  {eliminiere Band}
            ta[mx] := ta[k2]; ta[k2] := ta[k1];
            k1 := k1-1; k2 := k2-1
         end else
         if buf.key > f[ta[mx]]↑.key then
         begin
            tx := ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx;
            k2 := k2-1
         end
      until k2 = 0;
      if j < n then j := j+1 else j := nh+1
   until k1 = 0;
   for i := 1 to nh do
      begin tx := t[i]; t[i] := t[i+nh]; t[i+nh] := tx
      end
until l = 1;
reset(f[t[1]]); list(f[t[1]], t[1]);
```

```
    {geordnete Ausgabe ist auf t[1]}
end {tapemergesort} ;

begin {generiere File f0 mit Zufallswerten}
    leng := 200; rand := 7789; rewrite(f0);
    repeat rand := (131071*rand) mod 2147483647;
        buf.key := rand div 2147484; write(f0, buf);
        leng := leng - 1
    until leng = 0;
    reset(f0); list(f0,1);
    tapemergesort;
end .
```

Programm 2.15: Ausgeglichenes Mischsortieren

2.3.4. Mehrphasen-Sortieren

Wir haben nun die notwendigen Techniken erörtert und die eigentliche Grundlage zur Untersuchung und Programmierung eines weiteren Sortier-Algorithmus erworben, dessen Leistung die des ausgeglichenen Sortierens übertrifft. Wir haben gesehen, dass ausgeglichenes Mischen die reinen Kopieroperationen durch Zusammenfassen von Verteilen und Mischen in einer Phase eliminiert. Es erhebt sich die Frage, ob die gegebenen Bänder noch besser ausgenutzt werden könnten. Dies ist tatsächlich der Fall; der Schlüssel zu dieser nächsten Verbesserung liegt in der Aufgabe des strengen Begriffs des eigentlichen Durchlaufs, d.h. in der Aufgabe des Prinzips, stets gleich viele Ziel- und Quellen-Bänder zu verwenden und diese am Ende jedes Durchlaufs zu vertauschen. Statt dessen wird der Begriff eines Durchlaufs abgeschwächt. Diese Methode wurde von R.L. Gilstad [8] eingeführt und Mehrphasen-Sortierung (Polyphase Sort) genannt.

Sie wird zuerst an einem Beispiel mit drei Bändern illustriert. Zu jedem Zeitpunkt werden Elemente von zwei Bändern auf ein drittes Band gemischt. Jedesmal, wenn eines der Quellen-Bänder erschöpft ist, wird es sofort Ziel-Band der Operationen des Mischens von dem noch nicht abgearbeiteten Band und dem bisherigen Ziel-Band.

Da wir wissen, dass n Läufe auf jedem Eingabe-Band in n Läufe auf

dem Ausgabe-Band umgeformt werden, brauchen wir nur die Zahl der auf jedem Band vorhandenen Läufe festzuhalten (anstatt vorliegende Schlüssel anzugeben). In Fig. 2.14 gehen wir von zwei Eingabe-Bändern f1 und f2 aus, die je 13 bzw. 8 Läufe enthalten.

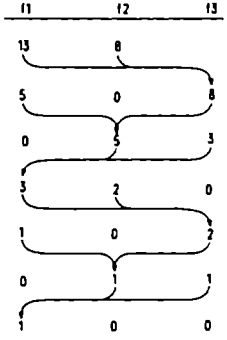


Fig. 2.14: Mischen mit drei Bändern

Somit werden im ersten "Durchlauf" 8 Läufe von f1 und f2 auf f3 gemischt, im zweiten "Durchlauf" werden die restlichen 5 Läufe von f3 und f1 auf f2 gemischt, usw. Zum Schluss ist f1 das geordnete File.

Ein zweites Beispiel zeigt die Mehrphasen-Methode mit sechs Bändern. Seien ursprünglich 16 Läufe auf f1, 15 auf f2, 14 auf f3, 12 auf f4 und 8 auf f5; beim ersten teilweisen Durchlauf werden 8 Läufe auf f6 geschrieben. Zum Schluss enthält nach Fig. 2.15 f2 die

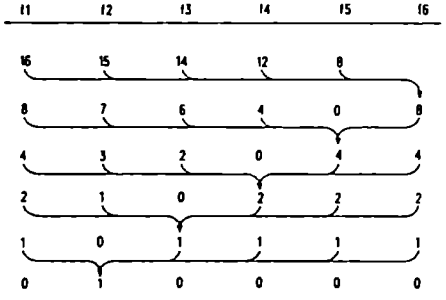


Fig. 2.15: Mischen mit 6 Bändern

geordnete Menge der Elemente.

Mehrphasen-Sortieren ist effizienter als ausgeglichenes Mischen, da es - für n gegebene Bänder - immer in einem (n-1)-Weg mischt statt in einem n/2-Weg. Da die Zahl der erforderlichen Durchläufe ungefähr logB N ist, wobei N die Zahl der zu sortierenden Elemente und B der Grad der Mischoperationen ist, verspricht Mehrphasen-Sortierung eine wesentliche Verbesserung gegenüber ausgeglichenem Mischen.

Natürlich war in obigem Beispiel die ursprüngliche Verteilung der Läufe sorgfältig gewählt. Um herauszufinden, welche ursprüngliche Verteilung der Läufe zu einem sauberen Funktionieren führt, arbeiten wir rückwärts, ausgehend von der Endverteilung wie in der letzten Zeile in Fig. 2.15 . Erstellen wir die Tabellen der beiden Beispiele neu und rotieren dabei jede Zeile bezüglich der voranstehenden um eine Position, so erhalten wir Tabellen 2.13 und 2.14 für sechs Durchläufe und drei bzw. sechs Bänder.

l	a[1](l)	a[2](l)	Summe a[i](l)
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13

Tabelle 2.13

l	a[1](l)	a[2](l)	a[3](l)	a[4](l)	a[5](l)	Summe a[i](l)
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Tabelle 2.14

Aus Tabelle 2.13 lassen sich folgende Relationen ableiten

$$a[2](l+1) = a[1](l)$$
$$a[1](l+1) = a[1](l) + a[2](l) \qquad \text{für } l >= 1 \qquad (2.40)$$

wobei $a[1](0) = 1$, $a[2](0) = 0$. Setzt man $a[1](l) = f[i]$, so erhalten wir

$$f[i+1] = f[i] + f[i-1] \text{ , für } i >= 1 \qquad (2.41)$$
$$f[1] = 1$$
$$f[0] = 0$$

Dies sind die rekursiven Regeln (oder Rekursions-Relationen) zur Definition der sogenannten Fibonacci-Zahlen:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Jede Fibonacci-Zahl ist die Summe ihrer beiden Vorgänger. Folglich müssen die Zahlen der ursprünglichen Läufe auf den beiden Bändern zwei aufeinanderfolgende Fibonacci-Zahlen sein, damit die Mehrphasen-Sortierung mit drei Bändern sauber abläuft.

Wie steht es mit dem zweiten Beispiel in Tabelle 2.14 mit sechs Bändern? Die Bildungsregeln lassen sich leicht herleiten als

$$a[5](l+1) = a[1](l) \qquad (2.42)$$
$$a[4](l+1) = a[1](l) + a[5](l) = a[1](l) + a[1](l+1)$$
$$a[3](l+1) = a[1](l) + a[4](l) = a[1](l) + a[1](l+1) + a[1](l-2)$$
$$a[2](l+1) = a[1](l) + a[3](l)$$
$$= a[1](l) + a[1](l-1)\ a[1](l-2) + a[1](l-3)$$
$$a[1](l+1) = a[1](l) + a[2](l)$$
$$= a[1](l-1) + a[1](l-2) + a[1](l-3) + a[1](l-4)$$

Setzt man $f[i]$ für $a[1](i)$, so gilt

$$f[i+1] = f[i] + f[i-1] + f[i-2] + f[i-3] + f[i-4], \text{ für } i>=4$$
$$f[4] = 1 \qquad (2.43)$$
$$f[i] = 0, \text{ für } i < 4$$

Dies sind die sogenannten Fibonacci-Zahlen der Ordnung 4. Allgemein werden die Fibonacci-Zahlen der Ordnung p wie folgt definiert.

$$f[i+1](p) = f[i](p) + f[i-1](p) + \ldots + f[n-p](p) \text{ , für } i >= p$$
$$f[p](p) = 1 \qquad (2.44)$$
$$f[i](p) = 0 \text{ , für } 0 <= i < p$$

Man beachte, dass die gewöhnlichen Fibonacci-Zahlen die der Ordnung 1 sind.

Wir haben nun gesehen, dass die ursprüngliche Zahl der Läufe für eine perfekte Mehrphasen-Sortierung mit n Bändern die Summe irgendwelcher n-1, n-2, ... , 1 (vgl. Tabelle 2.15) aufeinanderfolgender Fibonacci-Zahlen der Ordnung n-2 sind. Daraus folgt aber, dass

l	n= 3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001
15	1597	13745	34513	56417	76806	95617
16	2584	25281	66526	110913	152351	190465
17	4181	46499	128233	218049	302201	379399
18	6765	85525	247177	428673	599441	755749
19	10946	157305	476449	842749	1189041	1505425
20	17711	289329	918385	1656801	2358561	2998753

Zahl der ursprünglichen Läufe Summe f[i](l), die beim Mehrphasen-Sortieren mit n Bändern in l Phasen gemischt werden.

Tabelle 2.15

diese Methode nur für Eingaben anwendbar ist, deren Zahl von Läufen die Summe von n-1 solchen Fibonacci-Summen ist. Es stellt sich die wichtige Frage: Was tut man, wenn die Zahl der ursprünglichen Läufe keine solche ideale Summe ist?

Die Antwort ist sehr einfach (und für solche Situationen typisch): Wir simulieren die Existenz zusätzlicher leerer Läufe, so dass die Summe der echten und simulierten Läufe eine perfekte Summe bildet. Die leeren Läufe heissen Pseudo-Läufe.

Dies ist jedoch keine wirklich befriedigende Antwort, da sich sofort eine weitere und schwierigere Frage stellt: Wie erkennen wir diese Pseudo-Läufe während des Mischens? Bevor wir diese Frage beantworten, müssen wir zuerst das Problem der ursprünglichen Verteilung der Läufe untersuchen und uns für eine Regel zur Verteilung der wirklichen und der Pseudo-Läufe auf die n-1 Bänder entscheiden.

Um eine geeignete Regel für die Verteilung zu finden, müssen wir aber wissen, wie wirkliche und Pseudo-Läufe zu mischen sind. Natürlich bedeutet die Wahl eines Pseudo-Laufs vom Band i genau, dass das Band i während dieses Mischvorgangs zu ignorieren ist, so dass von weniger als n-1 Quellen gemischt wird. Mischen eines Pseudo-Laufs von allen n-1 Quellen-Bändern bedeutet keine echte Mischoperation, sondern das Aufzeichnen eines resultierenden Pseudo-Laufs auf dem Ausgabe-Band. Daraus schliessen wir, dass Pseudo-Läufe so gleichmässig wie möglich auf die n-1 Bänder zu verteilen sind, da wir bei wirklichen Mischvorgängen an möglichst vielen Quellen-Bändern interessiert sind.

Vergessen wir für einen Moment die Pseudo-Läufe, und betrachten wir das Problem der Verteilung einer unbekannten Zahl von Läufen auf n-1 Bänder. Es ist klar, dass die Fibonacci-Zahlen der Ordnung n-2, die die gewünschten Zahlen von Läufen auf jedem Band angeben, während des Verteilungsprozesses erzeugt werden können. Nehmen wir z.B. n = 6 an und schauen zurück auf Tabelle 2.14, so beginnen wir die Verteilung der Läufe, wie es in der Zeile mit dem Index l = 1 angegeben ist, (1, 1, 1, 1, 1); sind mehr Läufe vorhanden, so gehen wir weiter zur zweiten Zeile (2, 2, 2, 2, 1); ist die Quelle immer noch nicht erschöpft, so geht die Verteilung entsprechend der

dritten Zeile weiter (4, 4, 4, 3, 2), usw. Wir nennen den Zeilenindex Stufe. Je grösser natürlich die Zahl der Läufe ist, um so höher wird die Stufe der Fibonacci-Zahlen sein, die übrigens gleich der Zahl der für das anschliessende Sortieren notwendigen Mischdurchläufe oder Umschaltungen der Bänder ist.

Der Verteilungs-Algorithmus kann nun in einer ersten Version folgendermassen formuliert werden:

1. Das Ziel der Verteilung seien die Fibonacci-Zahlen der Ordnung n-2 und der Stufe 1.

2. Man verteile entsprechend dem gesetzten Ziel.

3. Ist das Ziel erreicht, berechne man die nächste Stufe der Fibonacci-Zahlen; die Differenz zwischen diesen und denjenigen der vorhergehenden Stufe bilden das neue Ziel der Verteilung. Man gehe zurück zu Schritt 2. Ist das Ziel nicht erreicht, da die Quelle erschöpft ist, beende man den Verteilungsprozess.

Die Definition der Fibonacci-Zahlen in (2.44) enthält die Regeln zur Berechnung der Zahlen der nächsten Stufe. Wir können unser Augenmerk daher auf Schritt 2 richten, in dem, entsprechend einem gegebenen Ziel, die nachfolgenden Läufe einer nach dem anderen auf die n-1 Bänder verteilt werden. Hier erscheinen nun die Pseudo-Läufe wieder in unseren Betrachtungen.

Nehmen wir an, dass wir beim Weiterschreiten zur nächsten Stufe das neue Ziel in den Differenzen d[i] mit i = 1...n-1 festhalten, wobei d[i] die Zahl der in diesem Schritt auf das Band i zu schreibenden Läufe angibt. Wir können nun annehmen, dass wir sofort d[i] Pseudo-Läufe auf das Band i "schreiben", und betrachten dann die nachfolgende Verteilung als Ersetzen der Pseudo-Läufe durch wirkliche Läufe, wobei wir jedesmal das Ersetzen durch Verkleinern der Zahl d[i] um 1 ausdrücken. Somit gibt d[i] die Zahl der Pseudo-Läufe auf Band i an, wenn die Quelle erschöpft ist.

Es ist nicht bekannt, welcher Algorithmus die beste Verteilung liefert, aber die folgende Methode hat sich als sehr gut erwiesen. Sie heisst horizontale Verteilung (vgl. [17], Seite 270). Diese

Bezeichnung ist verständlich aus der Betrachtung der in Silos aufgestapelten Läufe, wie es Fig. 2.16 für n=6 und Stufe 5 zeigt (vgl. Tabelle 2.14).

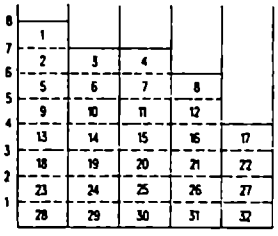


Fig. 2.16: Horizontale Verteilung beim Mehrphasen-Sortieren

Um eine Verteilung der restlichen Pseudo-Läufe so schnell wie möglich zu erreichen, muss das Austauschen durch wirkliche Läufe, das die Grösse der Stapel durch Herausnehmen auf horizontalen Ebenen verkleinert, von links nach rechts vorgenommen werden. Auf diese Art werden die Läufe so auf die Bänder verteilt, wie es die Zahlen ihrer Reihenfolge in Fig. 2.16 angeben.

Wir sind nun in der Lage, den Algorithmus in einer Prozedur selecttape zu beschreiben, die jedesmal nach Kopieren eines Laufs aufgerufen wird, wenn ein neues Band für den nächsten Lauf zu wählen ist. Eine Variable j bezeichne den Index des gegenwärtigen Ziel-Bandes. a[i] und d[i] bezeichnen die Zahlen der idealen Verteilung und der Pseudo-Läufe von Band i.

```
j: tapeno;                                                    (2.45)
a,d: array [tapeno] of integer;
level: integer
```

Diese Variablen werden auf folgende Werte initialisiert:

a[i] = 1 , d[i] = 1 für i = 1...n-1
a[n] = 0 , d[n] = 0 (leer)
j = 1 , level = 1 .

Man beachte, dass selecttape die nächste Zeile von Tabelle 2.14, d.h. die Werte a[1](l)...a[n-1](l), bei jedem Wechsel auf eine höhere Stufe zu berechnen hat. Das "nächste Ziel", d.h. die

Differenzen a[i] = a[i](l) - a[i](l-1) werden gleichzeitig berechnet. Der angegebene Algorithmus beruht auf der Tatsache, dass die resultierenden d[i] mit zunehmendem Index abnehmen (absteigende Treppe in Fig. 2.16). (Man beachte die Ausnahme beim Uebergang von Stufe 0 zu Stufe 1; dieser Algorithmus ist deshalb ab Stufe 1 zu verwenden.) Selecttape schliesst mit dem Verkleinern von d[j] um 1: diese Operation bedeutet das nachfolgende Ersetzen eines Pseudo-Laufs auf Band j durch einen wirklichen Lauf.

```
procedure selecttape;                                          (2.46)
   var i: tapeno; z: integer;
begin
   if d[j] < d[j+1] then j := j+1 else
   begin if d[j] = 0 then
      begin level := level + 1; z := a[1];
         for i := 1 to n-1 do
         begin d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
         end
      end;
      j := 1
   end;
   d[j] := d[j] - 1
end
```

Nehmen wir an, dass eine Routine zum Kopieren eines Laufs von der Quelle f0 auf f[j] zur Verfügung steht, so können wir die Phase der ursprünglichen Verteilung folgendermassen formulieren (immer unter der Annahme, dass die Quelle mindestens einen Lauf enthält):

```
repeat selecttape; copyrun                                     (2.47)
until eof(f0)
```

Hier müssen wir aber einen Moment innehalten, um uns an einen Effekt zu erinnern, der bei der Verteilung der Läufe nach dem bereits erörterten Algorithmus des natürlichen Mischens auftrat: Zwei nacheinander auf das gleiche Ziel-Band geschriebene Läufe können zu einem Lauf zusammenfallen und bewirken, dass die angenommene Zahl von Läufen falsch ist. Wählt man den Algorithmus zur Sortierung so, dass seine Korrektheit nicht von der Zahl der Läufe abhängt, so kann dieser Seiteneffekt ruhig ignoriert werden.

Bei der Mehrphasen-Sortierung sind wir aber hauptsächlich mit dem Aufzeichnen der genauen Zahlen der Läufe auf jedem Band beschäftigt. Folglich können wir den Effekt eines zufälligen Mischens nicht unbeachtet lassen.

Eine zusätzliche Komplikation des Verteilungs-Algorithmus ist daher nicht zu umgehen. Wir müssen die Schlüssel der letzten Elemente des letzten Laufs auf jedem Band aufbewahren. Dazu führen wir eine weitere Variable ein:

```
last: array [tapeno] of integer
```

Ein nächster Versuch, den Verteilungs-Algorithmus zu beschreiben, ist

```
repeat selecttape;                                          (2.48)
   if last[j] <= f0↑.key then
      "setzte den letzten Lauf fort";
   copyrun: last[j] := f0↑.key
until eof(f0)
```

Offensichtlich wurde dabei vergessen, dass last[j] erst nach dem Kopieren des ersten Laufs einen (definierten) Wert hat! Eine richtige Lösung verteilt zuerst je einen Lauf auf jedes der n-1 Bänder, ohne last[j] zu verwenden. Die restlichen Läufe werden nach Rezept (2.49) verteilt.

```
while not eof(f0) do                                        (2.49)
begin selecttape;
   if last[j] <= f0↑.key then
   begin {setze den letzten Lauf fort}
      copyrun;
      if eof(f0) then d[j] := d[j] + 1 else copyrun
   end
   else copyrun
end
```

Dabei nimmt man an, dass die Zuweisung von last[j] in der Prozedur copyrun vorgenommen wird.

Nun sind wir endlich in der Lage, den Haupt-Algorithmus für das Mehrphasen-Mischsortieren in Angriff zu nehmen. Seine wesentliche

Struktur ist dem Hauptteil des n-Weg-Mischprogramms ähnlich: eine äussere Schleife, deren Rumpf Läufe bis zur Erschöpfung der Quellen mischt, eine innere Schleife, deren Rumpf einen einzelnen Lauf von jedem Quellen-Band mischt, und eine innerste Schleife, deren Rumpf den ursprünglichen Schlüssel wählt und das betreffende Element auf das Ziel-Band bringt. Es gibt vier wesentliche Unterschiede:

1. Statt n/2 gibt es in jedem Durchlauf nur ein Ausgabe-Band.

2. Anstatt n/2 Eingabe- und n/2 Ausgabe-Bänder nach jedem Durchlauf umzuschalten, werden die Bänder rotiert. Dies geschieht wiederum durch Verwendung einer Bandindex-Abbildung t.

3. Die Zahl der Eingabe-Bänder ändert sich von Lauf zu Lauf; zu Beginn eines jeden Laufs wird sie aus den Zählern d[i] der Pseudo-Läufe bestimmt. Ist d[i] > 0 für alle i, so werden n-1 Pseudo-Läufe zu einem Pseudo-Lauf "gemischt", indem man nur den Zähler d[n] des Ausgabe-Bandes erhöht. Sonst wird ein Lauf von allen Bändern mit d[i] = 0 gemischt, und für alle übrigen Bänder wird d[i] verkleinert, um anzugeben, dass ein Pseudo-Lauf entfernt wurde. Wir bezeichnen die Zahl der bei einem Lauf beteiligten Eingabe-Bänder mit k.

4. Es ist unmöglich, die Beendigung einer Phase aus dem end-of-file Zustand des (n-1)-ten Bandes abzuleiten, da weitere Mischvorgänge mit Pseudo-Läufen von diesem Band nötig sein könnten. Statt dessen wird die theoretisch notwendige Zahl von Läufen aus den Koeffizienten a[i] bestimmt. Die Koeffizienten a[i](1) wurden während der Verteilungsphase berechnet und können nun wieder "rückwärts" ermittelt werden.

Der Hauptteil des Mehrphasen-Sortierens kann nun entsprechend diesen Regeln formuliert werden unter der Voraussetzung, dass alle n-1 Bänder mit ursprünglichen Läufen zurückgespult sind und dass die Bandabbildung auf t[i] = i initialisiert wurde.

```
repeat {mische von t[1] ... t[n-1] auf t[n]}                    (2.50)
    z := a[n-1]; d[n] := 0;
    repeat k := 0; {mische einen Lauf}
        {bestimme Zahl k der aktiven Eingabe-Läufe}
```

```
        for i := 1 to n-1 do
        if d[i] > 0 then d[i] := d[i]-1 else
           begin k := k+1; ta[k] := t[i]
           end;
        if k = 0 then d[n] := d[n] + 1 else
           "mische einen echten Lauf von t[1] ... t[k]"
        z := z-1
     until z = 0;
     reset(f[t[n]]);
     "rotiere Bänder in Abbildung t;
      berechne a[i] der nächsten Stufe";
     rewrite(f[t[n]]);  level := level - 1
  until level = 0;
  {geordnete Ausgabe ist auf t[1]}
```

Die eigentliche Mischoperation ist mit dem Programm des n-Weg-Mischsortierens nahezu identisch. Es unterscheidet sich nur durch einen etwas einfacheren Algorithmus zum Ausschalten eines Bandes. Die Rotation der Bandindex-Abbildung und der entsprechenden Zähler d[i] (sowie die Rückwärts-Berechnung der Koeffizienten a[i]) ergibt sich direkt. Die Details können aus Programm 2.16 entnommen werden, das den ganzen Algorithmus des Mehrphasen-Sortierens enthält.

```
program polysort(output);
{Mehrphasen-Sortieren mit n Bändern}
const n = 6;                    {Zahl der Bänder}
type item = record
               key: integer
            end ;
   tape = file of item;
   tapeno = 1..n;
var leng, rand: integer;        {für die Erstellung des Files}
   eot: Boolean;
   buf: item;
   f0: tape;  {f0 ist Eingabe-Band mit Zufallswerten}
   f: array [1..n] of tape;

procedure list(var f: tape; n: tapeno);
   var z: integer;
begin z := 0;
```

```
    writeln(' tape', n:2);
    while not eof(f) do
    begin read(f, buf); write(output, buf.key: 5); z := z+1;
       if z = 25 then
          begin writeln(output); z := 0
          end ;
    end ;
    if z <> 0 then writeln(output); reset(f)
end {list} ;

procedure polyphasesort;
    var i,j,mx,tn: tapeno;
       k, level: integer;
       a, d: array [tapeno] of integer;
            {a[j] = ideale Zahl von Läufen auf Band j}
            {d[j] = Zahl der Pseudo-Läufe auf Band j}
       dn, x, min, z: integer;
       last: array [tapeno] of integer;
         {last[j] = Schlüssel des letzten Elementes auf Band j}
       t,ta: array [tapeno] of tapeno;
            {Abbildung der Bandnummern}

    procedure selecttape;
       var i: tapeno; z: integer;
    begin
       if d[j] < d[j+1] then j := j+1 else
       begin if d[j] = 0 then
             begin level := level + 1; z := a[1];
                for i := 1 to n-1 do
                begin
                   d[i] := z + a[i+1] - a[i];
                   a[i] := z + a[i+1]
                end
             end :
             j := 1
       end ;
       d[j] := d[j] -1
    end ;
```

```
   procedure copyrun;
   begin {kopiere einen Lauf von f0 auf Band j}
      repeat read(f0, buf); write(f[j], buf)
      until eof(f0)  or  (buf.key > f0↑.key);
      last[j] := buf.key
   end ;

begin {verteile ursprüngliche Läufe}
   for i := 1 to n-1 do
      begin a[i] := 1; d[i] := 1; rewrite(f[i])
      end ;
   level := 1; j := 1; a[n] := 0; d[n] := 0;
   repeat selecttape; copyrun
   until eof(f0) or (j=n-1);
   repeat selecttape;
      if last[j] <= f0↑.key then
      begin {setze letzten Lauf fort}
         copyrun;
         if eof(f0) then d[j] := d[j] + 1 else copyrun
      end
      else copyrun
   until eof(f0);
   for i := 1 to n-1 do reset(f[i]);
   for i := 1 to n do t[i] := i;
   repeat {mische von t[1] ... t[n-1] auf t[n]}
      z := a[n-1]; d[n] := 0; rewrite(f[t[n]]);
      writeln(' level', level:4, '  tape', t[n]:4);
      for i := 1 to n do writeln(t[i], a[i], d[i]);
      repeat k := 0;  {mische einen Lauf}
         for i := 1 to n-1 do
         if d[i]  > 0 then d[i] := d[i]-1 else
            begin k := k+1; ta[k] := t[i]
            end ;
         if k = 0 then d[n] := d[n] + 1 else
         begin {mische einen wirklichen Lauf von t[1]...t[k]}
            repeat i := 1; mx := 1;
               min := f[ta[1]]↑.key;
               while i < k do
               begin i := i+1; x := f[ta[i]]↑.key;
                  if x < min then
```

```
                    begin min := x; mx := i
                    end
                end ;
                {ta[mx] enthält kleinstes Element,
                 bringe es auf t[n]}
                read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
                write(f[t[n]], buf);
                if (buf.key > f[ta[mx]]T.key) or eot then
                begin {nehme dieses Band weg}
                    ta[mx] := ta[k]; k := k-1
                end
            until k = 0
        end ;
        z := z-1
    until z = 0;
    reset(f[t[n]]); list(f[t[n]], t[n]);    {rotiere Bänder}
    tn := t[n]; dn := d[n]; z := a[n-1];
    for i := n downto 2 do
    begin t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
    end ;
    t[1] := tn; d[1] := dn; a[1] := z;
    {geordnete Ausgabe ist auf t[1]}
    level := level - 1
  until level = 0
end {polyphasesort} ;

begin {erzeuge File mit Zufallswerten}
  leng := 200; rand := 7789;
  repeat rand := (131071*rand) mod 2147483647;
    buf.key := rand div 2147484; write(f0, buf);
    leng := leng - 1
  until leng = 0;
  reset(f0); list(f0,1);
  polyphasesort
end .
```

Programm 2.16

2.3.5. Verteilung der ursprünglichen Läufe

Wir kamen zu den aufwendigeren sequentiellen Sortierprogrammen, da die einfacheren, auf Arrays arbeitenden Methoden von der Verfügbarkeit eines willkürlich zugreifbaren Speichers abhängen, der hinreichend gross sein muss, um die ganze Menge der zu sortierenden Daten zu enthalten. Sehr oft steht ein solcher Speicher nicht zur Verfügung; statt dessen müssen hinreichend grosse sequentielle Speichermedien wie Bänder verwendet werden. Es ist zu beachten, dass die bisher entwickelten sequentiellen Sortiermethoden praktisch keinerlei Vordergrundspeicher benötigen, mit Ausnahme für die File-Puffer und natürlich für das Programm selbst. Tatsächlich haben aber selbst kleine Rechenanlagen mehr willkürlich zugreifbaren Vordergrundspeicher, als die hier entwickelten Programme benötigen. Diesen nicht optimal einzusetzen, lässt sich nicht rechtfertigen.

Die Lösung liegt in der Kombination von Array- und File-Sortiertechniken. Insbesondere kann ein angepasstes Array-Sortierverfahren in der Verteilungsphase der ursprünglichen Läufe verwendet werden, so dass dann diese Läufe ungefähr eine dem verfügbaren Vordergrundspeicher entsprechende Länge l haben. Es ist klar, dass in den nachfolgenden Mischdurchläufen die Leistung durch zusätzliche Array-Sortierung nicht verbessert werden kann, da die Länge der Läufe stetig wächst und somit stets grösser ist als der verfügbare Hauptspeicher. Folglich können wir unsere Aufmerksamkeit auf die Verbesserung des Algorithmus zur Erzeugung der ursprünglichen Läufe konzentrieren.

Natürlich richten wir unser Augenmerk sofort auf die logarithmischen Methoden des Array-Sortierens. Am besten eignet sich die Methode des Baum-Sortierens oder Heapsort (vgl. 2.2.5). Der Heap kann als Kanal betrachtet werden, den alle File-Komponenten durchlaufen müssen, einige schneller, einige langsamer. Der kleinste Schlüssel wird sofort von der Spitze des Heap genommen und durch einen sehr effizienten Vorgang ersetzt. Das Durchsickern einer Komponente des Eingabe-Bandes f0 durch einen vollen "Heap-Kanal" h auf ein Ausgabe-Band f[j] lässt sich einfach beschreiben als

```
write(f[j],h[1]);                                    (2.51)
read(f0,h[1]);
sift(1,n)
```

"Sift" ist der in Abschnitt 2.2.5 beschriebene Vorgang für das Einordnen der neu eingefügten Komponente h[1] an ihren richtigen Platz. Man beachte, dass h[1] das kleinste Element des Heap ist. Fig. 2.17 enthält ein Beispiel hierfür.

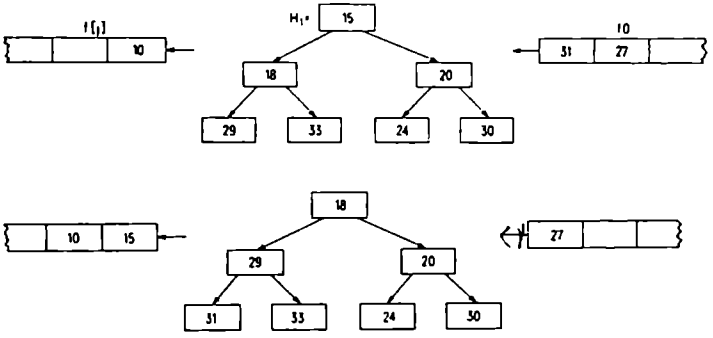


Fig. 2.17: Zustand vor und nach dem "Durchlauf" eines Elementes

Bei der genauen Ausarbeitung wird das Programm allerdings wesentlich komplexer, da

1. der Heap h ursprünglich leer ist und erst gefüllt werden muss,

2. gegen Schluss der Heap nur noch teilweise gefüllt bleibt, bis er schliesslich ganz leer wird,

3. wir den Anfang neuer Läufe erkennen müssen, um den Index j des Ausgabe-Bandes rechtzeitig umzuschalten.

Bevor wir weiterfahren, wollen wir die beteiligten Variablen formal definieren:

```
var f0: tape;                                          (2.52)
    f : array [tapeno] of tape;
    h : array [1..m] of item;
    l,r: integer
```

Dabei bezeichnet m die Grösse des Heap h. Zur Bezeichnung von m/2 verwenden wir die Konstante mh; l und r sind Indizes zu h. Der Prozess des Durchsickerns kann nun in fünf verschiedene Teile gegliedert werden.

1. Man lese die ersten mh Elemente von f0 und setze sie in die obere Hälfte des Heap ein, ohne Berücksichtigung einer Ordnung zwischen den Schlüsseln.

2. Man lese weitere mh Elemente und bringe sie in die untere Hälfte des Heap, indem man jedes Element an die entsprechende Position einordnet (Aufbauen des Heap).

3. Man initialisiere l und m und wiederhole den folgenden Schritt für alle restlichen Elemente auf f0:

   Man bringe h[1] auf das entsprechende Ausgabe-Band. Ist sein Schlüssel kleiner oder gleich dem Schlüssel des nächsten Elementes auf dem Eingabe-Band, so gehört dieses nächste Element zum gleichen Lauf und kann an seinen entsprechenden Platz eingefügt werden. Andernfalls reduziere man die Grösse des Heap und setze das neue Element in einen zweiten, "oberen" Heap, der für den nächsten Lauf aufgebaut wird. Wir geben die Grenze zwischen den beiden Heaps durch den Index l an. Somit besteht der "untere" oder gegenwärtige Heap aus den Elementen h[1]...h[l], der "obere" oder nächste Heap aus h[l+1] ... h[m]. Bei l = 0 wechsle man das Ausgabe-Band und setze l wieder auf m.

4. Nun ist die Quelle erschöpft. Man setze zunächst r auf m, übertrage dann den unteren Teil als Abschluss des gegenwärtigen Laufs. Gleichzeitig ist der obere Teil aufzubauen und schrittweise in die Positionen h[l+1] ... h[r] zu bringen.

5. Der letzte Lauf wird aus den restlichen Elementen des Heap generiert.

Wir sind nun in der Lage, die fünf Schritte in allen Einzelheiten als vollständiges Programm zu beschreiben, wobei wir jedesmal eine Prozedur selecttape am Ende jedes Laufs aufrufen und veranlassen, dass der Index des Ausgabe-Bandes geändert wird. In Programm 2.17 wird statt dessen eine Pseudo-Routine aufgerufen. Sie zählt nur die erzeugten Läufe. Alle Elemente werden auf Band f1 geschrieben.

```
program distribute(f0,f1,output);
{ursprüngliche Verteilung der Läufe durch Heapsort}
const  m = 30;  mh = 15;  {Grösse des Heap}
type item = record key: integer
            end ;
   tape = file of item;
   index = 0..m;
var l,r: index;
   f0,f1: tape;
   count: integer;   {Zähler der Läufe}
   h: array [1..m] of item;  {Heap}

procedure selecttape;
begin count := count + 1;
   {Leer. Zählt nur verteilte Läufe}
end {selecttape} ;

procedure sift(l,r: index);
   label 13;
   var i,j: integer; x: item;
begin i := l; j := 2*i; x := h[i];
   while j <= r do
   begin if j < r then
            if h[j].key > h[j+1].key then j := j+1;
         if x.key <= h[j].key then goto 13;
         h[i] := h[j]; i := j; j := 2*i
   end ;
13: h[i] := x
end ;

begin {erzeuge ursprüngliche Läufe durch Heapsort}
   count := 0; reset(f0); rewrite(f1); selecttape;
 {Schritt 1: FÜlle obere Hälfte des Heap h}
```

```
    l := m;
    repeat read(f0, h[1]); l := l-1
    until l = mh;
  {Schritt 2: Fülle die untere Hälfte des Heap h}
    repeat read(f0, h[1]); sift(l,m); l := l-1
    until l = 0;
  {Schritt 3: Bringe die Läufe durch den vollen Heap}
    l := m;
    while not eof(f0) do
    begin write(f1, h[1]);
       if h[1].key <= f0↑.key then
       begin {neuer Record gehört zum gleichen Lauf}
          read(f0, h[1]); sift(1,l);
       end else
       begin {neuer Record gehört zum nächsten Lauf}
          h[1] := h[l]; sift(1,l-1);
          read(f0, h[1]); if l <= mh then sift(l,m); l := l-1;
          if l = 0 then
          begin {Heap ist voll; beginne neuen Lauf}
             l := m; selecttape
          end
       end
    end ;
  {Schritt 4: Leere unteren Teil des Heap}
    r := m;
    repeat write(f1, h[1]);
       h[1] := h[l]; sift(1,l-1);
       h[l] := h[r]; r := r-1;
       if l <= mh then sift(l,r);  l := l-1
    until l = 0;
  {Schritt 5: Leere oberen Teil des Heap, erzeuge letzten Lauf}
    selecttape;
    while r > 0 do
    begin write(f1, h[1]);
       h[1] := h[r]; sift(1,r); r := r-1
    end ;
    writeln(count)
end .
```

Programm 2.17

Versuchen wir nun, in dieses Programm z.B. das Mehrphasen-Sortieren einzubeziehen, so stossen wir auf eine ernsthafte Schwierigkeit. Sie ergibt sich aus folgenden Umständen. Das Sortierprogramm besteht in seinem Anfangsteil aus einer ziemlich komplizierten Routine für die Umschaltung der Bänder und geht aus von der Verfügbarkeit einer Prozedur copyrun, die genau einen Lauf auf das gewählte Band bringt. Das Programm des Heap-Sortierens andererseits ist eine komplexe Routine, die sich auf die Verfügbarkeit einer abgeschlossenen Prozedur selecttape stützt, die nur ein neues Band wählt. Es gäbe keine Schwierigkeiten, wenn in einem (oder beiden) Programm(en) die entsprechende Prozedur nur an einer einzigen Stelle aufgerufen würde; sie werden aber in beiden Programmen an verschiedenen Stellen aufgerufen.

Diese Situation wird am besten wiedergegeben bei Verwendung einer sogenannten Coroutine; sie eignet sich für Fälle, in denen verschiedene Prozesse nebeneinanderher existieren. Das typische Beispiel ist die Kombination eines Prozesses, der einen Strom von Information in unterscheidbaren Einheiten erzeugt, und eines Prozesses, der diesen Strom verarbeitet. Diese Erzeuger-Verbraucher-Relation kann durch zwei Coroutinen ausgedrückt werden, wovon eine durchaus das Hauptprogramm selbst sein kann.

Die Coroutine kann als Prozedur oder Subroutine mit einem oder mehreren Schnittpunkten betrachtet werden. Stösst man auf einen solchen Schnittpunkt, so geht die Kontrolle zu dem Programm zurück, das die Coroutine aufgerufen hat. Wird die Coroutine erneut aufgerufen, so wird an diesem Schnittpunkt wieder eingesetzt. In unserem Beispiel könnten wir das Mehrphasen-Sortieren als Hauptprogramm betrachten, das das als Coroutine formulierte copyrun aufruft. Es besteht aus dem Hauptteil von Programm 2.17, in dem jeder Aufruf von selecttape nur einen Schnittpunkt darstellt. Der Test auf das Ende eines Files wäre dann systematisch zu ersetzen durch einen Test, ob die Coroutine ihr Ende erreicht hat. Eine logische Formulierung wäre eoc(copyrun) statt eof(f0).

Analyse und Folgerungen: Welche Leistung kann man von einem Mehrphasen-Sortieren mit ursprünglicher Verteilung der Läufe durch Heapsort erwarten? Untersuchen wir zuerst die durch die Einführung des Heap zu erwartenden Verbesserungen.

In einer Sequenz mit zufällig verteilten Schlüsseln ist die zu erwartende mittlere Länge der Läufe gleich 2. Wie gross ist die entsprechende Länge, nachdem die Sequenz durch einen Heap der Länge m gesickert ist? Man ist versucht, m zu sagen, glücklicherweise aber ist das Ergebnis nach der Wahrscheinlichkeitsrechnung viel besser, nämlich 2*m (vgl. [18], Seite 254). Der zu erwartende Verbesserungsfaktor ist also m.

Die Leistung des Mehrphasen-Sortierens kann aus Tabelle 2.15 abgeschätzt werden, die die maximale Zahl der ursprünglichen Läufe angibt, die in einer gegebenen Zahl von Teildurchläufen (Stufen) mit einer gegebenen Zahl n von Bändern sortiert werden kann. So kann z.B. mit n = 6 Bändern und einem Heap der Grösse m = 100 ein File mit bis zu 165'680'100 ursprünglichen Läufen in 20 Teildurchläufen sortiert werden. Dies ist eine beachtliche Leistung.

Betrachten wir nochmals die Kombination des Mehrphasen- und Heap-Sortierens, so können wir nur über die Komplexität dieses Programms staunen. Eigentlich löst es die gleiche, leicht zu definierende Aufgabe des Ordnens einer Menge von Elementen, wie es jedes der kurzen Programme tut, die auf den Prinzipien des direkten Array-Sortierens aufgebaut sind. Folgende zwei Punkte können als Schlussfolgerung aus diesem ganzen Kapitel betrachtet werden:

1. Die enge Verbindung zwischen Algorithmus und zugrunde liegender Datenstruktur und speziell der Einfluss der Datenrepräsentation auf die Wahl des Algorithmus.

2. Die Perfektion, mit der die Leistung eines Programms verbessert werden kann, selbst wenn die verfügbare Struktur für die Daten (Sequenz statt Array) für die Aufgabe ziemlich ungeeignet ist.

Uebungen

2.1. Welche der durch die Programme 2.1 bis 2.6, 2.8, 2.10 und 2.13 gegebenen Algorithmen sind stabile Sortiermethoden?

2.2. Würde Programm 2.2 immer noch korrekt laufen, wenn in der **while**-Klausel l <= r durch l < r ersetzt würde? Wäre es immer noch korrekt, wenn die Anweisungen r := m-1 und l := m+1 zu r := m und l := m vereinfacht würden? Falls nicht, suche man eine Menge von Werten a[1]...a[n], für die das geänderte Programm falsch wäre.

2.3. Man programmiere und messe die Ausführungszeit der drei Methoden des direkten Sortierens auf der eigenen Rechenanlage und bestimme Gewichte, mit denen die Faktoren C und M zu multiplizieren sind, damit sie echte Zeitabschätzungen ergeben.

2.4. Man teste das Programm 2.8 des Heapsort mit verschiedenen zufällig gewählten Eingabe-Sequenzen und bestimme, wie oft im Mittel die Anweisung "**goto** 13" ausgeführt wird. Da diese Zahl relativ klein ist, gewinnt folgende Frage an Bedeutung: Gibt es einen Weg, um den Test

```
x.key >= a[j].key
```

aus der while-Schleife herauszunehmen?

2.5. Man betrachte die folgende "auf der Hand liegende" Version des Zerlegungsprogramms 2.9:

```
i := 1; j := n; x := a[(n+1) div 2].key;
repeat
   while a[i].key < x do i := i+1;
   while x < a[j].key do j := j-1;
   w := a[i]; a[i] := a[j]; a[j] := w
until i > j
```

Man bestimme Mengen von Werten a[1]...a[n], für die diese Version falsch ist.

2.6. Man schreibe ein Programm, das die Algorithmen des Quicksort und Bubblesort folgendermassen kombiniert: Man verwende Quicksort zum Erstellen von (unsortierten) Zerlegungen der Länge m (1 <= m <= n) und dann Bubblesort zur Beendigung der Aufgabe. Man beachte, dass letzterer über den ganzen Array der n Elemente laufen kann und somit den Aufwand der "Buchhaltung" minimalisiert. Man bestimme den Wert von m, für den die gesamte Zeit des Sortierens minimal wird.

Bemerkung: Natürlich wird der beste Wert von m sehr klein sein. Es kann sich daher auszahlen, Bubblesort genau m-1 Mal auf den Array anzuwenden, statt einen letzten Durchlauf anzuschliessen, der nur feststellt, dass kein weiterer Austausch notwendig ist.

2.7. Man führe das gleiche Experiment wie in Aufgabe 2.6 mit direkter Auswahl statt mit Bubblesort durch. Selbstverständlich kann das Auswahlsortieren nicht über den ganzen Array stattfinden; deshalb ist ein grösserer Aufwand bei der Indizierung zu erwarten.

2.8. Man schreibe einen rekursiven Algorithmus für Quicksort entsprechend der Empfehlung, den kürzeren Teil der Zerlegung vor dem längeren Teil zu sortieren. Man löse den ersten Teil der Aufgabe in einer iterativen Anweisung, den zweiten durch rekursiven Aufruf. (Die Sortierprozedur wird somit nur einen rekursiven Aufruf enthalten anstelle der zwei in Programm 2.10 und keinem in Programm 2.11.)

2.9. Man finde eine Permutation der Schlüssel 1, 2, ... , n, für die Quicksort sich am schlechtesten (besten) erweist (n = 5,6,8).

2.10. Man konstruiere ein Programm für natürliches Mischen entsprechend dem Programm 2.13 für direktes Mischen, das auf einem Array doppelter Länge von beiden Enden her nach innen arbeitet, und vergleiche die Leistung mit der von Programm 2.13.

2.11. Man beachte, dass wir in einem natürlichen (2-Weg) Mischen

nicht blind den kleinsten unter den verfügbaren Schlüsseln wählen. Statt dessen wird beim Auftreten des Endes eines Laufs der Rest des anderen Laufs einfach auf die Ausgabe-Sequenz kopiert. Zum Beispiel führt Mischen von

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

zu der Sequenz

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

statt zu

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

wo die kleinsten Schlüssel weiter nach vorn gerutscht sind. Welcher Grund führt zu dieser Strategie?

2.12. Welchen Zweck erfüllt die Variable ta in Programm 2.15? Unter welchen Umständen wird die Anweisung

```
begin rewrite(f[ta[mx]]); ...
```

ausgeführt und wann die Anweisung

```
begin tx := ta[mx]; ... ?
```

2.13. Warum brauchen wir die Variable last im Mehrphasen-Sortierprogramm 2.16, nicht aber in Programm 2.15?

2.14. Eine dem Mehrphasen-Sortieren ähnliche Methode heisst Kaskaden-Mischsortieren ([17], S. 289 und [5]). Sie verwendet ein anderes Mischverfahren. Seien z.B. sechs Bänder T1, ..., T6 gegeben, so beginnt das Kaskaden-Mischen auch mit einer "perfekten Verteilung" der Läufe auf T1...T5, führt dann ein 5-Weg-Mischen von T1... T5 auf T6 aus, bis T5 leer ist, dann (ohne Verwendung von T6) ein 4-Weg-Mischen auf T5, dann ein 3-Weg-Mischen auf T4, ein 2-Weg-Mischen auf T3 und schliesslich ein Kopieren von T1 auf T2. Der nächste Durchlauf arbeitet auf die gleiche Art, beginnend mit einem 5-Weg-Mischen auf T1, usw. . Obwohl dieses Schema dem Mehrphasen-Sortieren unterlegen zu sein scheint, da es manchmal einige Bänder nicht verwendet und ausserdem einfache Kopieroperationen vorkommen, ist es überraschenderweise dem Mehrphasen-Sortieren für (sehr) grosse Files und sechs oder mehr Bänder überlegen. Man schreibe ein strukturiertes Programm für das Prinzip des Kaskaden-Mischens.

3. Rekursive Algorithmen

3.1. Einleitung

Ein Objekt heisst rekursiv, wenn es sich selbst teilweise enthält oder durch sich selbst definiert ist. Rekursion kommt nicht nur in der Mathematik, sondern auch im täglichen Leben vor. Wer hat etwa noch nie Reklamebilder gesehen, die sich selbst enthalten?

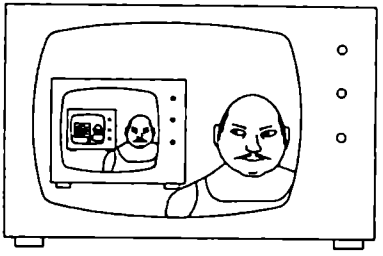
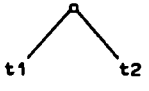


Fig. 3.1: Rekursion im Bild

Rekursion kommt speziell in mathematischen Definitionen zur Geltung. Bekannte Beispiele sind die natürlichen Zahlen, Baumstrukturen und gewisse Funktionen:

1. Natürliche Zahlen:
   (a) 1 ist eine natürliche Zahl.
   (b) Der Nachfolger einer natürlichen Zahl ist wieder eine natürliche Zahl.

2. Baumstrukturen:
   (a) ∅ ist ein Baum (genannt leerer Baum).
   (b) Wenn t1 und t2 Bäume sind, dann ist auch



ein Baum (auf dem Kopf stehend).

3. Die Fakultät n! (für nicht negative ganze Zahlen):
   (a) 0! = 1.
   (b) Wenn n>0, dann n! = n*(n-1)! .

Das Wesentliche der Rekursion ist die Möglichkeit, eine unendliche Menge von Objekten durch eine endliche Aussage zu definieren. Auf die gleiche Art kann eine unendliche Zahl von Berechnungen durch ein endliches rekursives Programm beschrieben werden, ohne dass das Programm explizite Schleifen enthält. Rekursive Algorithmen sind hauptsächlich dort angebracht, wo das Problem, die Funktion oder die Datenstruktur bereits rekursiv definiert sind. Ein rekursives Programm P kann im allgemeinen als Zusammensetzung $\underline{P}$ der Grundanweisungen S und P selbst ausgedrückt werden.

$$P = \underline{P}[S,P] \qquad (3.1)$$

Ein notwendiges und hinreichendes Werkzeug zur Darstellung rekursiver Programme ist die Prozedur oder Subroutine. Sie erlaubt, eine Anweisung mit einem Namen zu versehen, durch den die Anweisung aufgerufen werden kann. Enthält eine Prozedur P einen expliziten Aufruf von sich selbst, so heisst P direkt rekursiv; enthält P einen Aufruf einer zweiten Prozedur Q, die dann ihrerseits P (direkt oder indirekt) aufruft, so heisst P indirekt rekursiv. Das Vorhandensein von Rekursion muss daher nicht direkt aus dem Programmtext ersichtlich sein.

Es ist üblich, mit einer Prozedur eine Menge lokaler Objekte zu verknüpfen, z.B. können Variable, Konstanten, Typen und Prozeduren lokal definiert sein und ausserhalb dieser Prozedur weder Existenz noch Bedeutung haben. Bei jeder rekursiven Aktivierung einer solchen Prozedur wird ein Satz lokaler, gebundener Variablen kreiert. Obwohl sie dieselben Namen haben wie die Objekte in der lokalen Menge des vorangehenden Aufrufs der Prozedur, so haben sie doch im allgemeinen verschiedene Werte. Namenskonflikte werden durch die Regeln für den Gültigkeitsbereich von Namen ausgeschlossen: die Namen beziehen sich immer auf die zuletzt geschaffene Menge von Variablen. Die gleiche Regel gilt auch für Prozedur-Parameter, die durch die Definition an die Prozedur gebunden werden.

Wie Wiederholungsanweisungen bergen auch rekursive Prozeduren die Gefahr nicht abbrechender Berechnungen und verlangen daher die Betrachtung des Problems der Termination. Grundlegend ist sicher die Bedingung, dass der rekursive Aufruf einer Prozedur abhängt von einer Bedingung B, die irgendwann nicht mehr erfüllt ist. Das Schema für rekursive Algorithmen kann nun genauer beschrieben werden als

$$P = \underline{\text{if}}\ B\ \underline{\text{then}}\ P[S,P] \tag{3.2}$$

oder

$$P = P[S,\ \underline{\text{if}}\ B\ \underline{\text{then}}\ P] \tag{3.3}$$

Zum Nachweis der Termination einer Schleife verwendet man die grundlegende Technik der Definition einer Funktion $f(\underline{x})$ ($\underline{x}$ ist die Menge der Programm-Variablen), so dass $f(\underline{x}) <= 0$ die Abbruchbedingung (der while- oder repeat-Klausel) impliziert. Man beweist dann, dass $f(\underline{x})$ bei jeder Wiederholung abnimmt. Auf die gleiche Art kann Termination eines rekursiven Programms bewiesen werden, indem man zeigt, dass jede Ausführung von P den Wert $f(\underline{x})$ verkleinert. Eine besonders einfache Art, Termination sicherzustellen, ist die Zuordnung eines (value) Parameters, nennen wir ihn n, zu P. Der rekursive Aufruf von P mit dem Parameterwert n-1 garantiert dann das strikte Abnehmen des Wertes $f(\underline{x})$, und Ersetzen der Bedingung B durch $n>0$ impliziert somit Termination. Dies kann durch die folgenden Programmschemata ausgedrückt werden.

$$P(n) = \underline{\text{if}}\ n > 0\ \underline{\text{then}}\ P[S,P(n-1)] \tag{3.4}$$

$$P(n) = P[S,\ \underline{\text{if}}\ n > 0\ \underline{\text{then}}\ P(n-1)] \tag{3.5}$$

Für praktische Anwendungen ist es wesentlich zu zeigen, dass die grösste Tiefe der Rekursion nicht nur endlich, sondern sogar klein ist. Dies ist bedingt durch den Speicherplatz, den jede rekursive Aktivierung einer Prozedur P für ihre lokalen Variablen benötigt. Zusätzlich zu diesen Variablen muss der gegenwärtige Stand der Berechnung gespeichert werden, damit er nach Beendigung des erneuten Aufrufs von P für die Fortsetzung der vorangegangenen Aktivierung wieder zur Verfügung steht. Wir haben diese Situation schon bei der Entwicklung der Prozedur Quicksort in Kapitel 2 angetroffen. Beim "naiven" Erstellen des Programms aus einer

Anweisung, die die n Elemente in zwei Teile zerlegt und diese Teile durch zwei rekursive Aufrufe sortiert, konnte sich die Tiefe der Rekursion im schlimmsten Fall n nähern. Durch geschicktes Ueberarbeiten der Situation war es möglich, diese Tiefe auf log n zu beschränken. Die Differenz zwischen n und log n ist gross genug, um einen für Rekursion völlig ungeeigneten Fall in einen für Rekursion bestens geeigneten überzuführen.

3.2. Wo Rekursion zu vermeiden ist

Rekursive Algorithmen eignen sich besonders, wenn das zugrunde liegende Problem oder die zu behandelnden Daten rekursiv definiert sind. Das bedeutet aber nicht, dass eine solche rekursive Definition eine Garantie dafür bietet, dass ein rekursiver Algorithmus auch der beste Weg zur Lösung des Problems ist. Tatsächlich hat die Erklärung des Konzeptes rekursiver Algorithmen anhand von ungeeigneten Beispielen wesentlich zur Entstehung einer weitverbreiteten Abneigung und Antipathie gegen die Rekursion in der Programmierung beigetragen und zur Gleichsetzung von Rekursion mit Ineffizienz geführt. Dazu kommt noch die Tatsache, dass die weitverbreitete Programmiersprache FORTRAN den rekursiven Gebrauch von Subroutinen verbietet und so die Schaffung rekursiver Lösungen verhindert, selbst wenn sie angebracht wären.

Programme, bei denen auf rekursive Algorithmen zu verzichten ist, können durch die äquivalenten Schemata (3.6) und (3.7) charakterisiert werden, die die Gestalt ihrer Zusammensetzung aufzeigen.

$$P = \underline{\text{if}}\ B\ \underline{\text{then}}\ (S;\ P) \qquad (3.6)$$

$$P = (S;\ \underline{\text{if}}\ B\ \underline{\text{then}}\ P) \qquad (3.7)$$

Diese Schemata ergeben sich in den Fällen, in denen Werte nach einer einfachen Rekursionsrelation zu berechnen sind. Betrachten wir das bekannte Beispiel der Fakultäten f(i) = i! :

```
  i  = 0, 1, 2, 3,  4,   5, ...                    (3.8)
f(i) = 1, 1, 2, 6, 24, 120, ...
```

Die "nullte" Zahl ist explizit definiert als f(0) = 1, während die folgenden Zahlen üblicherweise - rekursiv - definiert sind unter Verwendung ihrer Vorgänger:

```
f(i+1) = (i+1)*f(i)                                (3.9)
```

Diese Formel legt einen rekursiven Algorithmus zur Berechnung der n-ten Fakultät nahe. Führen wir zwei Variablen I und F zur Bezeichnung der Werte i und f(i) auf der i-ten Stufe der Rekursion ein, so führen folgende Schritte der Berechnung zu den nächsten Zahlen der Folgen (3.8).

```
I := I+1; F := I*F                                 (3.10)
```

Ersetzt man in (3.6) S durch (3.10), so erhält man das rekursive Programm

```
P = if I < n then (I := I+1; F := I*F; P)          (3.11)
```

mit den Initialisierungsanweisungen

```
I := 0; F := 1; P
```

~~Die erste Zeile von~~ (3.11) lautet in unserer gebräuchlichen Programmiernotation

```
procedure P;                                       (3.12)
begin if I < n then
      begin I := I+1;  F := I*F;  P
      end
end
```

Häufiger, aber im wesentlich äquivalent, ist die Formulierung (3.13). P ist ersetzt durch eine sogenannte Funktions-Prozedur. Das ist eine Prozedur, mit der ein Resultat explizit verknüpft ist und die daher direkt als Teil eines Ausdrucks auftreten kann. Die Variable F wird deshalb überflüssig; die Rolle der Variablen I wird

durch den expliziten Prozedur-Parameter übernommen.

```
function F(I: integer): integer;                    (3.13)
begin if I > 0 then F := I*F(I-1)
               else F := 1
end
```

Es ist offensichtich, dass in diesem Fall die Rekursion durch eine einfache Wiederholung ersetzt werden kann, nämlich durch

```
I := 0; F := 1;                                     (3.14)
while I < n do
   begin I := I+1; F := I*F
   end
```

Im allgemeinen sollten Programme gemäss Schema (3.6) oder (3.7) auf die Form von Schema (3.15) umgeschrieben werden.

```
P  =  (x:=x0; while B do S)                         (3.15)
```

Es gibt auch kompliziertere Rekursionsrelationen, die auf iterative Form gebracht werden können, bzw. gebracht werden sollten. Ein Beispiel dafür ist die Berechnung der Fibonacci-Zahlen, die durch folgende Relation definiert sind:

```
fib(n+1) = fib(n) + fib(n-1)        für n>0         (3.16)
fib(1) = 1, fib(0) = 0
```

Ein erster naiver Versuch führt zu folgendem Programm:

```
function Fib(n: integer): integer;                  (3.17)
begin if n = 0 then Fib := 0 else
      if n = 1 then Fib := 1 else
      Fib := Fib(n-1) + Fib(n-2)
end
```

Die Berechnung von fib(n) durch den Aufruf Fib(n) führt zu rekursivem Aufruf der Funktions-Prozedur. Wieviele Aufrufe finden statt? Offenbar führt jeder Aufruf mit n>1 zu zwei weiteren Aufrufen, d.h. die Gesamtzahl der Aufrufe wächst exponentiell (vgl.

Fig. 3.2). Ein solches Programm ist offensichtlich unbrauchbar.

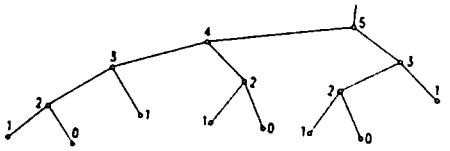


Fig. 3.2: Die 15 Aufrufe von Fib(5)

Die Fibonacci-Zahlen können jedoch sehr wohl nach einem Iterationsschema berechnet werden, das die wiederholte Berechnung gleicher Werte durch die Verwendung von Hilfsvariablen vermeidet. Für die Hilfsvariablen gilt die Beziehung x = fib(i) und y = fib(i-1).

```
[berechne x = fib(n) für n>0]                                    (3.16)
i := 1; x := 1; y := 0;
while i < n do
   begin z := x;  i := i+1;
         x := x+y;  y := z
   end
```

(Man beachte, dass die drei Zuweisungen an x, y und z durch zwei Zuweisungen ohne Verwendung der Hilfsvariablen z ausgedrückt werden können: x := x+y; y := x-y.)

Die Folgerung aus diesen Ueberlegungen ist, dass man auf Verwendung von Rekursion immer dann verzichten sollte, wenn es eine offensichtliche Lösung mit Iteration gibt. Das bedeutet aber nicht, dass Rekursion um jeden Preis zu umgehen ist. Es gibt viele gute Anwendungen für Rekursion, wie die folgenden Abschnitte zeigen werden. Die Tatsache, dass Implementationen von rekursiven Prozeduren auf nicht rekursiven Maschinen existieren, beweist, dass für praktische Zwecke jedes rekursive Programm in ein rein iteratives umgeformt werden kann. Das verlangt jedoch das explizite Verwalten eines Rekursions-Stacks, und diese Operationen verschleiern oft das Grundprinzip eines Programms so sehr, dass es schwer zu verstehen ist. Zusammenfassend lässt sich sagen, dass Algorithmen, die ihrem

Wesen nach eher rekursiv als iterativ sind, auch als rekursive Prozeduren formuliert werden sollten. (Der Leser möge dazu die Programme 2.10 und 2.11 vergleichen.)

Der Rest dieses Kapitels zeigt die Entwicklung rekursiver Programme in Fällen, in denen Rekursion angebracht ist. Auch in Kapitel 4 wird Rekursion häufig verwendet, wenn die zugrunde liegenden Datenstrukturen die Wahl einer rekursiven Lösung zweckmässig und natürlich erscheinen lassen.

3.3. Zwei Beispiele rekursiver Programme

Das attraktive Muster von Fig. 3.5 besteht aus sechs übereinandergezeichneten Kurven. Diese Kurven haben ein regelmässiges Muster, und es liegt nahe, sie mit einem Zeichengerät unter Rechnerkontrolle zu zeichnen. Unser Ziel ist es, das Rekursionsschema zu finden, nach dem das Zeichenprogramm erstellt werden kann. Eine Untersuchung ergibt, dass drei der überlagerten Kurven, nennen wir sie H(1), H(2) und H(3), die in Fig. 3.3 gezeigte Gestalt haben.

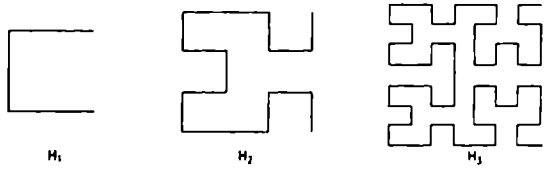


Fig. 3.3: Die Hilbert-Kurven H(1) - H(3)

Die Figuren zeigen, dass jede Kurve H(i+1) aus vier Einheiten von H(i) zusammengesetzt ist. Dazu wird die Grösse dieser Einheiten halbiert, sie werden geeignet rotiert und durch drei gerade Linien verbunden. Man kann sich H(1) zusammengesetzt denken aus vier Einheiten einer leeren Figur H(0), die durch drei Linien verbunden sind. H(i) heisst Hilbert-Kurve der Ordnung i nach ihrem Erfinder D. Hilbert (1891).

Nehmen wir an, wir hätten zum Zeichnen zwei Koordinaten-Variablen x und y, eine Prozedur setplot (die die Feder auf die Koordinaten x und y setzt) und eine Prozedur plot (die die Feder von der gegenwärtigen Position zu der von x und y angegebenen Position bewegt).

Da jede der Kurven H(i) aus vier Kopien in halber Grösse von H(i-1) besteht, ist es natürlich, die Prozedur, die H(i) zeichnen soll, zusammenzusetzen aus vier Teilen, von denen jeder H(i-1) in richtiger Grösse und in richtigem Winkel zeichnet. Bezeichnen wir die vier Teile mit A, B, C und D und die die Verbindungslinien zeichnenden Routinen durch Pfeile in entsprechender Richtung, dann erhalten wir folgendes Rekursionsschema (vgl. Fig. 3.3):

```
A: D ← A ↓ A → B                                   (3.19)
B: C ↑ B → B ↓ A
C: B → C ↑ C ← D
D: A ↓ D ← D ↑ C
```

Mit h als Länge der Einheitsstrecke läßt sich die dem Schema A entsprechende Prozedur leicht schreiben unter Verwendung rekursiver Aufrufe analoger Prozeduren entsprechend den Schemata B, D und A selbst.

```
procedure a(i: integer);                           (3.20)
begin if i > 0 then
      begin d(i-1); x := x-h; plot;
            a(i-1); y := y-h; plot;
            a(i-1); x := x+h; plot;
            b(i-1)
      end
end
```

Diese Prozedur wird vom Hauptprogramm einmal für jede zu überlagernde Hilbert-Kurve aufgerufen. Das Hauptprogramm bestimmt den Anfangspunkt der Kurve, d.h. die Anfangswerte von x und y und die Einheitslänge h. h0 bezeichnet die volle Breite des Blattes und genügt der Bedingung h0 = 2k für irgendein k >= n (siehe Fig. 3.4). Das ganze Programm zeichnet die n Hilbert-Kurven H(1) ... H(n) (siehe Programm 3.1 und Fig. 3.5).

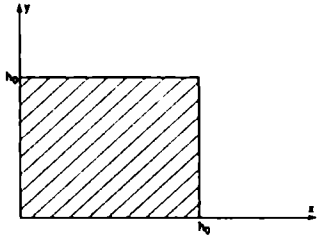


Fig. 3.4: Die Zeichenfläche

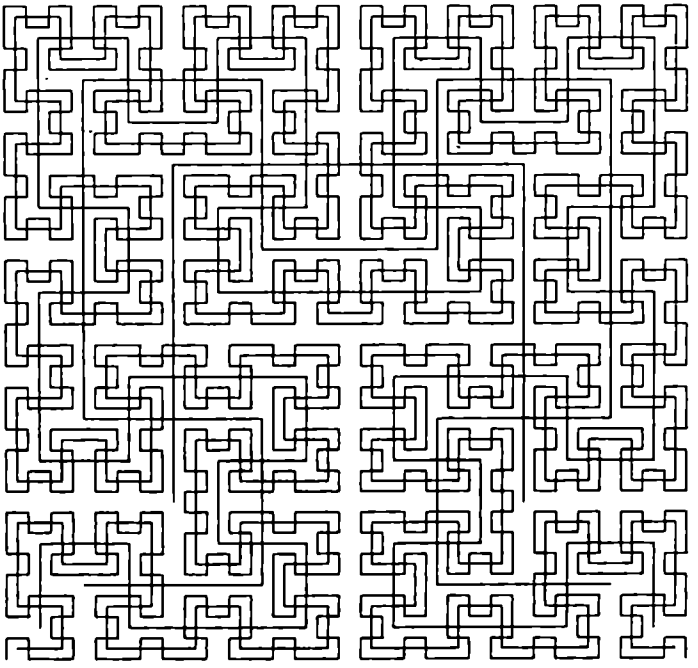


Fig 3.5: Hilbert-Kurven H(1) ... H(5)

```
program hilbert(pf,output);
{zeichne die Hilbert-Kurven der Ordnung 1 bis n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
    pf: file of integer;

procedure b(i: integer); forward;
procedure c(i: integer); forward;
procedure d(i: integer); forward;

procedure a(i: integer);
begin if i > 0 then
      begin d(i-1); x := x-h; plot;
            a(i-1); y := y-h; plot;
            a(i-1); x := x+h; plot;
            b(i-1)
      end
end ;

procedure b;
begin if i > 0 then
      begin c(i-1); y := y+h; plot;
            b(i-1); x := x+h; plot;
            b(i-1); y := y-h; plot;
            a(i-1)
      end
end ;

procedure c;
begin if i > 0 then
      begin b(i-1); x := x+h; plot;
            c(i-1); y := y+h; plot;
            c(i-1); x := x-h; plot;
            d(i-1)
      end
end ;

procedure d;
begin if i > 0 then
      begin a(i-1); y := y-h; plot;
```

```
            d(i-1); x := x-h; plot;
            d(i-1); y := y+h; plot;
            c(i-1)
        end
end :

begin startplot;
    i := 0; h := h0; x0 := h div 2 + h; y0 := h div 2;
    repeat {zeichne die Hilbert-Kurve der Ordnung i}
        i := i+1; h := h div 2;
        x0 := x0 + (h div 2); y0 := y0 + (h div 2);
        x := x0; y := y0; setplot;
        a(i)
    until i = n;
    endplot
end .
```

Programm 3.1: Hilbert-Kurven

Ein ähnliches, jedoch komplexeres und ästhetisch anspruchsvolleres Beispiel wird in Fig. 3.7 gezeigt. Dieses Muster entstand wiederum durch Ueberlagerung mehrerer Kurven; zwei davon zeigt Fig. 3.6. S(i) steht für Sierpinski-Kurve i-ter Ordnung.

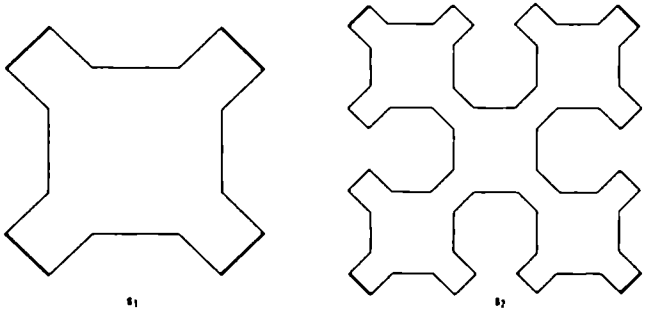


Fig. 3.6: Sierpinski-Kurven S(1) und S(2)

Welches ist das Rekursionsschema? Man ist versucht, das Blatt S(1) als eine grundlegende Figur zu isolieren, vielleicht unter Auslas-

sung einer Ecke. Dies führt jedoch zu keiner Lösung. Der wesentliche Unterschied zwischen Sierpinski- und Hilbert-Kurven liegt darin, dass die Sierpinski-Kurven geschlossen sind (ohne Ueberschneidungen). Das bedingt, dass das zugrunde liegende Rekursionsschema eine offene Kurve sein muss und dass die Verbindung der vier Teile nicht zum Rekursionsmuster selbst gehört. Diese vier Verbindungen erkennt man als die in Fig. 3.6 dick ausgezogenen Geraden in den vier äussersten "Ecken". Man kann sie sich vorstellen als Teil einer nicht leeren Anfangskurve S(0), bestehend aus einem auf der Spitze stehenden Quadrat.

Nun ist das Rekursionsschema leicht aufzustellen. Die vier grundlegenden Teile werden mit A, B, C und D bezeichnet, und das Zeichnen der Verbindungslinien wird explizit angegeben. Man beachte, dass die vier Teile bis auf Rotation um 90 Grad identisch sind.

Das Grundmuster der Sierpinski-Kurve ist

S: A ↘ B ↙ C ↖ D ↗ (3.21)

und die Rekursionsmuster sind

A: A ↘ B ⇒ D ↗ A (3.22)
B: B ↙ C ⇓ A ↘ B
C: C ↖ D ⇐ B ↙ C
D: D ↗ A ⇑ C ↖ D

(Doppelpfeile bezeichnen Strecken doppelter Einheitslänge.)

Mit den gleichen Zeichenoperationen wie beim Beispiel der Hilbert-Kurve kann man obenstehendes Rekursionsschema ohne Schwierigkeiten als (direkt und indirekt) rekursiven Algorithmus formulieren.

```
procedure a(i: integer);                                    (3.23)
begin if i > 0 then
      begin a(i-1);  x := x+h;  y := y-h;  plot;
            b(i-1);  x := x+2*h;  plot;
            d(i-1);  x:= x+h;  y := y+h;  plot;
            a(i-1)
      end
end
```

Die Prozedur ist aus der ersten Zeile des Schemas (3.22) abgeleitet. Die den Mustern B, C und D entsprechenden Prozeduren werden analog abgeleitet. Das Hauptprogramm entspricht dem Muster (3.21) und hat die Aufgabe, die Anfangswerte für die Koordinaten festzulegen und die Einheitslänge h entsprechend dem Papierformat zu wählen (siehe Programm 3.2).

```
program sierpinski(pf,output);
{Zeichne die Sierpinski-Kurven der Ordnung 1 bis n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
    pf: file of integer;     {Zeichen-File}

procedure b(i: integer); forward;
procedure c(i: integer); forward;
procedure d(i: integer); forward;


procedure a(i: integer);
begin if i > 0 then
      begin a(i-1); x := x+h; y := y-h; plot;
            b(i-1); x := x + 2*h; plot;
            d(i-1); x := x+h; y := y+h; plot;
            a(i-1)
      end
end ;

procedure b;
begin if i > 0 then
      begin b(i-1); x := x-h; y := y-h; plot;
            c(i-1); y := y - 2*h; plot;
            a(i-1); x := x+h; y := y-h; plot;
            b(i-1)
      end
end ;

procedure c;
begin if i > 0 then
      begin c(i-1); x := x-h; y := y+h; plot;
            d(i-1); x := x - 2*h; plot;
```

```
            b(i-1); x := x-h; y := y-h; plot;
            c(i-1)
      end
end ;

procedure d;
begin if i > 0 then
      begin d(i-1); x := x+h; y := y+h; plot;
            a(i-1); y := y + 2*h; plot;
            c(i-1); x := x-h; y := y+h; plot;
            d(i-1)
      end
end ;

begin startplot;
   i := 0; h := h0 div 4; x0 := 2*h; y0 := 3*h;
   repeat i := i+1; x0 := x0-h;
      h := h div 2; y0 := y0+h;
      x := x0; y := y0; setplot;
      a(i); x := x+h; y := y-h; plot;
      b(i); x := x-h; y := y-h; plot;
      c(i); x := x-h; y := y+h; plot;
      d(i); x := x+h; y := y+h; plot;
   until i = n;
   endplot
end .
```

Programm 3.2: Sierpinski-Kurven

Fig. 3.7 zeigt das Resultat dieses Programms für n = 4. Man beachte, dass S(0) nicht gezeichnet wurde.

Die Eleganz der Verwendung von Rekursion ist in diesen Beispielen offensichtlich und überzeugend. Die Richtigkeit der Programme kann leicht aus ihrer Struktur und ihrem Aufbaumuster hergeleitet werden. Ausserdem garantiert die Verwendung des expliziten Verschachtelungs-Parameters i nach Schema (3.5) die Termination, da die Tiefe der Rekursion nicht grösser werden kann als n.

Im Gegensatz zu dieser rekursiven Formulierung sind äquivalente

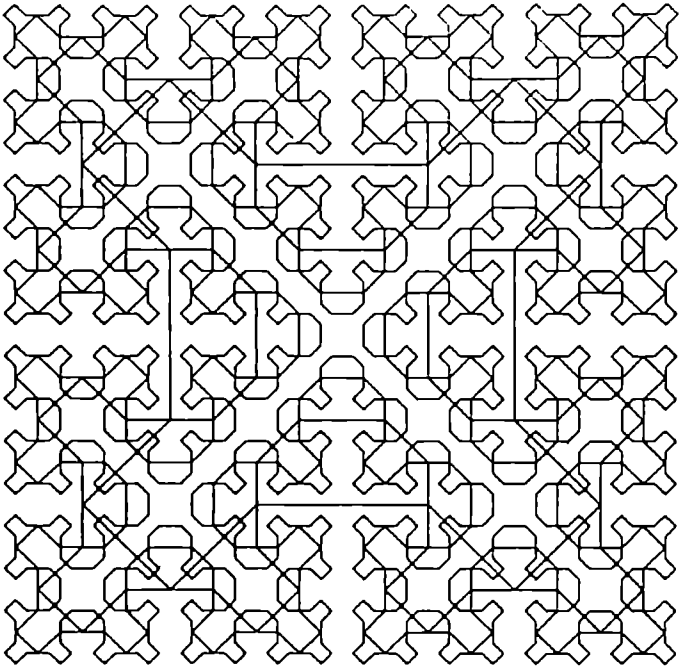


Fig. 3.7: Sierpinski-Kurven S(1) ... S(4)

Programme, die den expliziten Gebrauch von Rekursion vermeiden, äusserst schwerfällig und unklar. Dem Leser sei sehr empfohlen, sich von dieser Tatsache zu überzeugen, indem er die in [34] angegebenen Programme zu verstehen sucht.

3.4. Backtracking Algorithmen

Ein besonders fesselndes Ziel der Programmierung ist das Thema der allgemeinen Problemlösung. Es soll ein Algorithmus zum Finden von Lösungen spezieller Probleme bestimmt werden, und zwar nicht durch Befolgen einer festen Vorschrift für die Berechnung, sondern durch Versuchen und Nachprüfen (trial and error). Gewöhnlich wird der Prozess des Versuchens und Nachprüfens in einzelne Teilschritte

zerlegt. Oft lassen sich diese Schritte auf natürliche Art in rekursiver Form ausdrücken und bestehen aus der Untersuchung einer endlichen Zahl untergeordneter Schritte. Wir können allgemein den ganzen Prozess als einen Prozess des Versuchens und Nachprüfens sehen, der einen Baum von untergeordneten Problemen aufbaut und durchläuft. In vielen Problemen wächst dieser Suchbaum sehr schnell, gewöhnlich exponentiell in Abhängigkeit von einem gegebenen Parameter. Der Aufwand des Suchens wächst entsprechend. Häufig kann der Suchbaum nur durch heuristische Ueberlegungen beschnitten und die Berechnung damit auf vernünftige Grenzen reduziert werden.

Es ist nicht unsere Absicht, allgemeine heuristische Regeln in diesem Text zu erörtern, wir wollen vielmehr in diesem Kapitel das allgemeine Prinzip, nach dem die Aufgabe der Lösung solcher Probleme in untergeordnete Aufgaben unterteilt wird, sowie die Anwendung der Rekursion behandeln. Wir beginnen mit der Darstellung der zugrunde liegenden Technik anhand eines wohlbekannten Beispiels, nämlich dem Weg des Springers.

Gegeben sei ein n*n Brett. Ein Springer - der nach den Schachregeln bewegt werden kann - wird auf das Feld mit den Anfangskoordinaten xß,yß gestellt. Zu finden ist nun eine Ueberdeckung des ganzen Brettes, sofern dies möglich ist. Damit ist ein Weg von n**2-1 Zügen zu berechnen, so dass jedes Feld des Brettes genau einmal berührt wird.

Bei der Ueberdeckung der n*n Felder muss man herausfinden, welches der nächste Zug ist, bzw. ob ein Zug möglich ist. Wir wollen deshalb einen Algorithmus definieren, der versucht, den nächsten Zug auszuführen. (3.24) zeigt eine erste Näherung.

```
procedure "versuche nächsten Zug";                    (3.24)
begin "initialisiere Auswahlvorgang";
  repeat "wähle nächsten Kandidaten aus
          der Liste der nächsten Züge";
    if "annehmbar" then
    begin "zeichne Zug auf";
      if "Brett nicht voll" then
      begin "versuche nächsten Zug";
        if not "erfolgreich" then
```

```
                "lösche vorangehende Aufzeichnung"
            end
        end
    until "erfolgreich" or "keine Kandidaten mehr"
end
```

Wenn wir diesen Algorithmus genauer beschreiben wollen, müssen wir einige Entscheidungen über die Darstellung der Daten treffen. Ein naheliegender Schritt ist die Darstellung des Brettes durch eine Matrix, nennen wir sie H. Ausserdem wollen wir einen Typ für die Indexwerte einführen:

```
type index=1..n;                                        (3.25)
var H: array [index,index] of integer
```

Jedes Feld des Brettes wird durch eine ganze Zahl und nicht durch einen Booleschen Wert, der das Belegtsein angibt, dargestellt, da wir die Entwicklung der sukzessiven Belegung der Felder festhalten möchten. Naheliegend ist die Wahl:

```
H[x,y] = 0 : Feld <x,y> wurde noch nicht besucht        (3.26)
H[x,y] = i : Feld <x,y> wurde im i-ten Zug besucht.
             (1 <= i <= n*n)
```

Die nächste Entscheidung betrifft die Wahl geeigneter Parameter. Sie dienen zur Festlegung der Ausgangsbedingungen für den nächsten Zug und geben Auskunft über den Erfolg. Für den ersten Punkt eignet sich die Angabe der Koordinaten x und y, von denen aus der Zug zu machen ist, und der Nummer i des Zuges (zur Speicherung). Die letzte Aufgabe bedingt einen Booleschen Resultat-Parameter: q = true bedeutet Erfolg, q = false Misserfolg.

Welche Anweisungen können nun aufgrund dieser Entscheidungen verfeinert werden? Sicherlich kann "Brett nicht voll" ausgedrückt werden durch "i < n**2". Wenn wir zwei lokale Variable u und v für die Koordinaten des Zieles eines möglichen Zuges einführen, können wir ausserdem das Prädikat "annehmbar" ausdrücken durch die logische Kombination der Bedingungen, dass das neue Feld auf dem Brett liegt, d.h. 1<=u<=n und 1<=v<=n, und dass das Feld bisher noch nicht berührt wurde, d.h. H[u,v] = 0. Das Aufzeichnen des

legalen Zuges wird ausgedrückt durch die Zuweisung H[u,v]:= i und das Löschen dieser Aufzeichnung durch H[u,v]:= 0. Wird eine lokale Variable q1 eingeführt und als Resultat-Parameter beim rekursiven Aufruf dieses Algorithmus verwendet, dann kann q1 für "erfolgreich" stehen. Damit erhalten wir Formulierung (3.27).

```
procedure try(i: integer; x,y: index;                    (3.27)
              var q: Boolean);
   var u,v: integer; q1: Boolean;
begin "initialisiere Auswahlvorgang";
   repeat "u,v seien Kandidaten für Zug nach Schachregeln";
      if (1 <= u <= n) and (1 <= v <= n) and (H[u,v]=0) then
      begin H[u,v] := i;
         if i < sqr(n) then
         begin try(i+1,u,v,q1);
            if not q1 then H[u,v] := 0
         end else q1 := true
      end
   until q1 or "keine Kandidaten mehr";
   q := q1
end
```

Ein weiterer Schritt der Verfeinerung führt uns nun zu einem Programm, das vollständig in der Notation unserer Programmiersprache ausgedrückt ist. Es ist zu beachten, dass das Programm bis jetzt völlig unabhängig von den Regeln für die Züge des Springers entwickelt wurde. Die Betrachtung dieser Eigenheiten des Problems wurde absichtlich zurückgestellt. Aber jetzt ist es an der Zeit, sie zu berücksichtigen. Für ein gegebenes Paar <x,y> von Ausgangs-

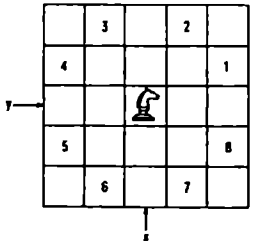


Fig. 3.8: Springer-Züge

koordinaten gibt es acht mögliche Zielkoordinaten. In Fig. 3.8 sind sie von 1 bis 8 durchnumeriert.

Man erhält u,v aus x,y einfach durch Addition der Differenz der Koordinaten, die entweder in einem Array der Differenzpaare oder in zwei Arrays einzelner Differenzen gespeichert sind. Diese Arrays seien mit a und b bezeichnet und geeignet initialisiert. Dann kann ein Index k verwendet werden, um den "nächsten Kandidaten" anzugeben. Einzelheiten sind in Programm 3.3 ausgeführt. Die rekursive Prozedur try wird gestartet durch einen Aufruf mit den Koordinaten x0,y0 des Feldes als Parameter, von dem aus der Weg beginnt. Diesem Feld ist der Wert 1 zuzuweisen, alle übrigen sind als frei zu kennzeichnen:

```
H[x0,y0]:= 1; try(2,x0,y0,q)
```

Ein weiteres Detail darf nicht übersehen werden. Eine Variable H[u,v] existiert nur, wenn beide, u und v, innerhalb der Array-Grenzen 1...n liegen. Folglich ist der Ausdruck in (3.27), der für "annehmbar" in (3.24) eingesetzt wurde, nur gültig, wenn die beiden ersten Bedingungen erfüllt sind. Eine saubere Formulierung wird in Programm 3.3 gezeigt, in dem ausserdem die doppelte Relation 1<=u<=n ersetzt wird durch den Ausdruck u in [1,2,...,n], der für hinreichend kleines n effizienter ist. Tabelle 3.1 enthält die Lösungen für die Startwerte <1,1>, <3,3> und n = 5 sowie für <1,1> und n = 6. Der Resultat-Parameter q und die lokale Variable q1 können zur Vereinfachung des Programms durch eine globale Variable ersetzt werden.

1	6	15	10	21		23	10	15	4	25
14	9	20	5	16		16	5	24	9	14
19	2	7	22	11		11	22	1	18	3
8	13	24	17	4		6	17	20	13	8
25	18	3	12	23		21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

Tabelle 3.1

```
program knightstour(output);
const n = 5; nsq = 25;
type index = 1..n;
var i,j: index;
    q: boolean;
    s: set of index;
    a,b: array [index] of integer;
    h: array [index, index] of integer;

procedure try(i: integer; x,y: index; var q: boolean);
   var k,u,v: integer; q1: boolean;
begin k := 0;
   repeat k := k+1; q1 := false;
      u := x + a[k]; v := y + b[k];
      if (u in s) and (v in s) then
      if h[u,v] = 0 then
      begin h[u,v] := i;
         if i < nsq then
            begin try(i+1,u,v,q1);
               if not q1 then h[u,v] := 0
            end else q1 := true
      end
   until q1 or (k=8);
   q := q1
end {try} ;

begin s := [1,2,3,4,5];
   a[1] :=  2; b[1] :=  1;
   a[2] :=  1; b[2] :=  2;
   a[3] := -1; b[3] :=  2;
   a[4] := -2; b[4] :=  1;
   a[5] := -2; b[5] := -1;
   a[6] := -1; b[6] := -2;
   a[7] :=  1; b[7] := -2;
   a[8] :=  2; b[8] := -1;
   for i := 1 to n do
      for j := 1 to n do h[i,j] := 0;
   h[1,1] := 1; try(2,1,1,q);
```

```
    if q then
       for i := 1 to n do
       begin for j := 1 to n do write(h[i,j]:5);
          writeln
       end
    else writeln(' keine Lösung ')
end .
```

Programm 3.3: Weg des Springers

Welche allgemeingültigen Regeln können nun aus diesem Beispiel abgeleitet werden? Welches Programm-Muster lässt es erkennen, das für diese Art von Algorithmen zur "Lösung von Problemen" typisch ist? Das charakteristische Merkmal ist, dass Schritte in Richtung Ziel versucht und aufgezeichnet werden, die man später wieder rückgängig macht und in den Aufzeichnungen löscht, sobald man erkannt hat, dass es ein Schritt in eine "Sackgasse" war. Diese Aktion heisst Backtracking. Das allgemeine Muster (3.28) ist aus der Prozedur (3.24) abgeleitet unter der Annahme, dass die Zahl der möglichen Kandidaten in jedem Schritt endlich ist.

```
procedure try;                                          (3.28)
begin "initialisiere Wahl der Kandidaten";
   repeat "wähle den nächsten";
      if "annehmbar" then
      begin "zeichne ihn auf";
         if "Lösung unvollständig" then
         begin "versuche nächsten Schritt";
            if not "erfolgreich" then "lösche Aufzeichnung"
         end
      end
   until "erfolgreich" or "keine weiteren Kandidaten mehr"
end
```

Echte Programme können natürlich verschiedene, aus (3.28) abgeleitete Formen annehmen. Ein häufig gefundenes Muster verwendet einen expliziten Stufen-Parameter, der die Tiefe der Rekursion angibt und der eine einfache Bedingung für die Termination erlaubt.

Wenn ausserdem bei jedem Schritt die Zahl der zu untersuchenden

Kandidaten gleich ist, etwa m, dann ist das abgeleitete Schema (3.29) angebracht. Es ist zu starten mit der Anweisung "try (1)".

```
procedure try(i: integer);                                    (3.29)
   var k: integer;
begin k := 0;
   repeat k := k+1; "wähle k-ten Kandidaten";
      if "annehmbar" then
      begin "zeichne ihn auf";
         if i < n then
         begin try(i+1);
            if not "erfolgreich" then "lösche Aufzeichnung"
         end
      end
   until "erfolgreich" or (k=m)
end
```

Im Rest des Kapitels werden drei weitere Beispiele behandelt. Sie zeigen verschiedene Verkörperungen des abstrakten Schemas (3.29) und sind zur weiteren Illustration geeigneter Verwendung von Rekursion angeführt.

3.4.1. Das Problem der acht Damen

Das Problem der acht Damen ist ein gängiges Beispiel für die Verwendung der Methode des Versuchens und Nachprüfens und für Backtracking-Algorithmen. Es wurde bereits 1850 von C.F. Gauss untersucht, jedoch nicht völlig gelöst. Dies sollte niemanden überraschen. Nach dem bisher Gesagten haben diese Probleme die charakteristische Eigenschaft, dass sie sich nicht analytisch lösen lassen. Tatsächlich erfordern sie einen grossen Aufwand an exakter Arbeit, Geduld und Zielsicherheit. Solche Algorithmen gewannen erst durch die automatische Rechenanlage an Bedeutung, da diese die benötigten Eigenschaften in einem wesentlich höheren Grad als Menschen und sogar Gelehrte besitzt. Das Problem der acht Damen ist folgendermassen formuliert (siehe auch [32]):

> Acht Damen sind auf einem Schachbrett so aufzustellen, dass keine Dame eine andere bedroht.

Verwenden wir Schema (3.29) als Schablone, so erhalten wir leicht folgenden ersten Entwurf einer Lösung:

```
procedure try(i: integer);                                    (3.30)
begin "initialisiere Wahl für i-te Dame"
   repeat "treffe nächste Wahl";
      if "sicher" then
      begin "setze die Dame";
         if i < 8 then
         begin try(i+1);
            if not "erfolgreich" then "entferne die Dame"
         end
      end
   until "erfolgreich" or "keine Positionen mehr"
end
```

Zur Fortsetzung müssen wir uns über die Darstellung der Daten in einigen Punkten festlegen. Von den Schachregeln wissen wir, dass eine Dame alle anderen Figuren bedroht, die auf dem Brett in der gleichen Kolonne, Zeile oder Diagonale stehen. Folglich kann jede Kolonne genau eine Dame enthalten, und die Wahl einer Position für die i-te Dame kann auf die i-te Kolonne beschränkt werden. Als Parameter i wird deshalb der Kolonnen-Index gewählt; der Auswahlprozess für die Positionen beschränkt sich dann auf die acht möglichen Werte für einen Zeilen-Index j.

Die Frage der Darstellung der acht Damen auf dem Brett bleibt offen. Naheliegend wäre wiederum eine quadratische Matrix zur Darstellung des Brettes. Aber schon kurzes Nachdenken zeigt, dass eine solche Repräsentation zu sehr umständlichen Operationen beim Testen der möglichen Positionen führt. Dies ist sehr unerwünscht, da es die am häufigsten ausgeführte Operation ist. Wir sollten deshalb die Repräsentation der Daten so wählen, dass das Prüfen so einfach wie möglich wird. Das beste Rezept ist, die wesentliche und häufig gebrauchte Information so direkt wie möglich darzustellen. In unserem Fall ist dies nicht die Position der Damen, sondern die Tatsache, ob bereits eine Dame auf einer Zeile und auf einer Diagonalen steht, in der wir eine weitere Dame plazieren wollen. (Wir wissen bereits, dass genau eine Dame in jeder Kolonne k steht für 1<=k<=i.)

Dies führt zur folgenden Wahl der Variablen:

```
var x: array [1..8] of integer;                    (3.31)
    a: array [1..8] of Boolean;
    b: array [b1..b2] of Boolean;
    c: array [c1..c2] of Boolean
```

wobei

x[i] die Position der Dame in der i-ten Kolonne bezeichnet;
a[j] bedeutet "keine Dame steht in der j-ten Zeile";
b[k] bedeutet "keine Dame besetzt die k-te /-Diagonale";
c[k] bedeutet "keine Dame sitzt in der k-ten \-Diagonalen".

Die Wahl für die Index-Grenzen b1, b2, c1 und c2 ist bestimmt durch die Art der Berechnung der Indizes b und c; wir sehen, dass in einer /-Diagonalen die Summe der Koordinaten i und j und in einer \-Diagonalen die Differenz i-j für alle Felder konstant ist. Programm 3.4 zeigt eine geeignete Lösung.

Mit diesen Daten kann die Anweisung "setze Dame" ausgeführt werden zu

```
x[i] := j; a[j] := false;                          (3.32)
b[i+j] := false; c[i-j] := false
```

und Anweisung "entferne die Dame" wird verfeinert zu

```
a[j] := true; b[i+j] := true; c[i-j] := true       (3.33)
```

Die Bedingung "sicher" ist erfüllt, wenn das Feld <i,j> in einer noch freien Zeile und in freien Diagonalen liegt (dargestellt durch "true"). Sie kann also ausgedrückt werden durch den logischen Durchschnitt

```
a[j] and b[i+j] and c[i-j]                         (3.34)
```

Dies vervollständigt die Entwicklung dieses Algorithmus, der als Ganzes in Programm 3.4 gezeigt wird. Die berechnete Lösung x = (1,5,8,6,3,7,2,4) ist in Fig. 3.9 gezeigt.

```
program eightqueen1(output);
{suche eine Lösung des Problems der acht Damen}
var i: integer; q: boolean;
   a: array [ 1.. 8] of boolean;
   b: array [ 2..16] of boolean;
   c: array [-7.. 7] of boolean;
   x: array [ 1.. 8] of integer;

procedure try(i: integer; var q: boolean);
   var j: integer;
begin j := 0;
   repeat j := j+1; q := false;
      if a[j] and b[i+j] and c[i-j] then
      begin x[i] := j;
         a[j] := false; b[i+j] := false; c[i-j] := false;
         if i < 8 then
         begin try(i+1,q);
            if not q then
            begin a[j] := true; b[i+j] := true; c[i-j] := true
            end
         end else q := true
      end
   until q or (j=8)
end {try} ;

begin
   for i :=  1 to  8 do a[i] := true;
   for i :=  2 to 16 do b[i] := true;
   for i := -7 to  7 do c[i] := true;
   try(1,q);
   if q then
      for i := 1 to 8 do write(x[i]:4);
   writeln
end .
```

Programm 3.4: Problem der acht Damen

Bevor wir die Umgebung des Schachbretts verlassen, soll das Beispiel des Problems der acht Damen zur Illustration einer wichtigen

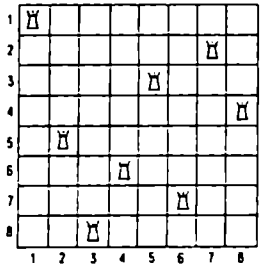


Fig. 3.9: Eine Lösung des Acht-Damen Problems

Erweiterung des Algorithmus des Versuchens und Nachprüfens dienen. Die Erweiterung besteht - allgemein gesagt - im Finden nicht nur einer, sondern aller Lösungen eines gestellten Problems.

Die Erweiterung lässt sich leicht anbringen. Erinnern wir uns daran, dass das Generieren der Kandidaten systematisch geschehen muss, um zu garantieren, dass kein Kandidat mehr als einmal generiert wird. Diese Eigenschaft des Algorithmus entspricht einem systematischen Durchsuchen des Baumes der Kandidaten, wobei jeder Knoten genau einmal inspiziert wird. Sie erlaubt - sobald eine Lösung einmal gefunden und aufgezeichnet ist - einfach mit dem nächsten Kandidaten weiterzufahren, der durch den systematischen Auswahlprozess bestimmt wird. Das allgemeine Schema ist abgeleitet aus (3.29) und dargestellt in (3.35).

```
procedure try(i: integer);                                    (3.35)
   var k: integer;
begin
   for k := 1 to m do
   begin "wähle k-ten Kandidaten";
      if "annehmbar" then
      begin "zeichne ihn auf";
         if i < n then try(i+1) else "drucke Lösung";
         "lösche Aufzeichnung"
      end
   end
end
```

Man beachte, dass die Vereinfachung der Terminations-Bedingung des Auswahlprozesses auf den einen Vergleich k = m die Anwendung einer for-Anweisung anstelle der repeat-Anweisung erlaubt. Die Suche nach allen möglichen Lösungen ist überraschenderweise mit einem einfacheren Programm möglich als die Suche nach einer einzigen Lösung.

Der erweiterte Algorithmus zur Bestimmung aller 92 Lösungen des Problems der acht Damen ist in Programm 3.5 gezeigt. Im wesentlichen gibt es aber nur zwölf signifikant verschiedene Lösungen; unser Programm erkennt keine Symmetrien. Die zwölf zuerst gefundenen Lösungen sind in Tabelle 3.2 aufgeführt. Die Zahlen N in der rechten Kolonne geben die Häufigkeit des Testens der Sicherheit eines Feldes an. Das Mittel dieser Zahlen für alle 92 Lösungen beträgt 161.

```
program eightqueens(output);
var i: integer;
    a: array [ 1.. 8] of boolean;
    b: array [ 2..16] of boolean;
    c: array [-7.. 7] of boolean;
    x: array [ 1.. 8] of integer;

procedure print;
   var k: integer;
begin for k := 1 to 8 do write(x[k]:4);
   writeln
end {print} ;

procedure try( i: integer);
   var j: integer;
begin
   for j := 1 to 8 do
      if a[ j] and b[ i+j] and c[ i-j] then
      begin x[i] := j;
         a[ j] := false; b[ i+j] := false; c[ i-j] := false;
         if i < 8 then try( i+1) else print;
         a[ j] := true; b[ i+j] := true; c[ i-j] := true
      end
end {try} ;
```

```
begin
   for i :=  1 to  8 do a[i] := true;
   for i :=  2 to 16 do b[i] := true;
   for i := -7 to  7 do c[i] := true;
   try(1)
end .
```

Programm 3.5: Acht Damen

x1	x2	x3	x4	x5	x6	x7	x8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

Tabelle 3.2: Zwölf Lösungen des Acht-Damen Problems

3.4.2. Das Problem der stabilen Heirat

Gegeben seien zwei verschiedene Mengen A und B mit gleicher Kardinalität. Gesucht ist eine Menge von n Paaren <a,b>, so dass a in A und b in B sowie einige Nebenbedingungen erfüllt sind. Es existieren verschiedene Kriterien für solche Paare, eines davon heisst "die Regel der stabilen Heirat":

Sei A eine Menge von Männern und B eine Menge von Frauen. Jeder Mann und jede Frau hat eine Reihenfolge der Bevorzugung ihrer Partner aufgestellt. Wenn die n Paare so gewählt werden, dass es einen Mann und eine Frau gibt, die nicht miteinander verheiratet sind, die sich aber gegenseitig vor ihren tatsächlichen Ehepartnern

den Vorzug geben, dann heisst die Wahl unstabil. Existiert kein solches Paar, so ist die Wahl stabil.

Diese Situation ist typisch für viele ähnliche Probleme, bei denen die Wahl je nach Bevorzugung zu treffen ist, wie z.B. die Wahl einer Schule durch Studenten, die Wahl der verschiedenen Zweige der Armee durch Rekruten, usw. Das Beispiel der Heiraten ist besonders anschaulich; zu beachten ist jedoch, dass die aufgestellte Liste der Bevorzugungen invariant ist und sich nicht ändert, nachdem eine bestimmte Zuordnung einmal getroffen wurde. Diese Regel vereinfacht das Problem, sie ist aber gleichzeitig eine Verzerrung der Wirklichkeit (genannt Abstraktion).

Ein Weg zur Suche einer Lösung ist der Versuch, Mitglieder aus beiden Mengen nacheinander zusammenzuführen, bis beide Mengen leer sind. Sollen alle stabilen Zuordnungen gefunden werden, so können wir eine Lösung leicht skizzieren unter Verwendung von Programmschema (3.35) als Schablone. Bezeichnet try(m) den Algorithmus zum Finden eines Partners für den Mann m, und erfolgt diese Suche in der Reihenfolge der Liste der Bevorzugungen des Mannes, so ergibt sich eine erste Version in (3.36).

```
procedure try(m: man);                                    (3.36)
   var r: rank;
begin
   for r := 1 to n do
   begin "greife r-te Bevorzugung des Mannes m heraus";
      if "annehmbar" then
      begin "notiere Heirat";
         if "m ist nicht letzter Mann" then try(succ(m))
            else "notiere stabile Menge";
         "annuliere Heirat"
      end
   end
end
```

Hier sind wir nun wieder an einem Punkt angelangt, an dem wir ohne weitere Entscheidungen über die Darstellung der Daten nicht weiterkommen. Wir führen drei skalare Typen ein, deren Werte der Einfachheit halber die ganzen Zahlen 1 bis n sind. Obowhl diese

drei Typen formal gleich sind, trägt die verschiedene Bezeichnung wesentlich zur Klarheit bei. Es wird dadurch offensichtlich, wozu eine Variable verwendet wird.

```
type man = 1..n;                                    (3.37)
     woman = 1..n;
     rank = 1..n
```

Die Anfangswerte sind durch zwei Matrizen dargestellt, die die Bevorzugungen der Männer und Frauen enthalten.

```
var wmr: array [man, rank] of woman                 (3.38)
    mwr: array [woman, rank] of man
```

Folglich bezeichnet wmr[m] die Liste der Bevorzugungen von Mann m, d.h. wmr[m][r] = wmr[m,r] ist die Frau, die auf dem r-ten Platz in der Liste des Mannes m figuriert. Entsprechend ist mwr[w] die Liste der Bevorzugungen der Frau w, und mwr[w,r] ist ihre r-te Wahl.

Das Ergebnis steht in einem Array x von Frauen, und x[m] bezeichnet die Partnerin des Mannes m. Um die Symmetrie zwischen Mann und Frau aufrechtzuerhalten - die oft zitierte "Gleichberechtigung" - wird ein zusätzlicher Array y eingeführt, so dass y[w] den Partner der Frau w angibt.

```
var x: array [man] of woman;                        (3.39)
    y: array [woman] of man
```

Es ist klar, dass y nicht unbedingt erforderlich ist, da es bereits in x vorhandene Information enthält. Es gelten folgende Beziehungen

$$x[y[w]] = w, \qquad y[x[m]] = m \tag{3.40}$$

für alle verheirateten m und w. Somit kann der Wert y[w] durch einfaches Durchsuchen von x erhalten werden; der Array y erhöht aber die Effizienz des Algorithmus. Die durch x und y dargestellte Information wird zur Bestimmung der Stabilität einer vorgelegten Menge von Heiraten benötigt. Da diese Menge schrittweise durch Verheiraten eines Paares und anschliessendes Testen der Stabilität konstruiert wird, werden x und y sogar benötigt, bevor alle ihre

Komponenten definiert sind. Um die bereits definierten Komponenten festzuhalten, können wir Boolesche Arrays einführen

```
singlem : array [man] of Boolean                    (3.41)
singlew : array [woman] of Boolean
```

wobei aus

```
not singlem[m]    folgt, dass x[m] definiert ist,
not singlew[w]    folgt, dass y[w] definiert ist.
```

Betrachtet man aber den vorgeschlagenen Algorithmus, so erkennt man schnell, dass der Zivilstand eines Mannes durch den Wert m auf einfache Art bestimmt ist, nämlich durch

```
not singlem[k] = k<m .                              (3.42)
```

Damit kann auf den Array singlem verzichtet werden, und der Name singlew wird vereinfacht zu single.

Diese Abmachungen ergeben die in (3.43) gezeigte Verfeinerung: das Prädikat "annehmbar" wird als Konjunktion single[w] and "stabil" ausgedrückt, wobei "stabil" eine weiter auszuarbeitende Funktion ist.

```
procedure try(m: man);                              (3.43)
   var r: rank;  w: woman;
begin for r:= 1 to n do
      begin w:= wmr[m,r];
            if single[w] and "stabil" then
            begin x[m] := w;  y[w] := m;  single[w] := false;
               if m < n then try(succ(m))
                        else "notiere stabile Menge";
               single[w] := true
            end
      end
end
```

An diesem Punkt ist die starke Aehnlichkeit dieser Lösung mit Programm 3.5 immer noch festzustellen, da beide aus demselben Schema hervorgegangen sind.

Die wesentliche Aufgabe ist nun die Verfeinerung des Algorithmus zur Bestimmung der Stabilität. Leider kann die Stabilität nicht durch einen solch einfachen Ausdruck dargestellt werden wie die Sicherheit der Damen in Programm 3.5. Zunächst sollte man beachten, dass sich die Stabilität nach Definition aus Vergleichen von Bevorzugungen oder Reihenfolgen ergibt. Die Reihenfolge von Männern oder Frauen ist aber in der bisherigen Aufstellung der Daten nicht explizit verfügbar. Sicherlich kann der Platz von der Frau w in der Einschätzung von Mann m berechnet werden, aber dies erfordert eine aufwendige Suche von w in wmr[m].

Da die Berechnung der Stabilität eine äusserst häufige Operation ist, soll diese Information direkt zur Verfügung gestellt werden. Dazu führen wir zwei Matrizen ein

```
rmw : array [man, woman] of rank;          (3.44)
rwm : array [woman, man] of rank
```

so dass rmw[m,w] den Platz der Frau w in der Liste der Bevorzugungen von Mann m bezeichnet und rwm[w,m] den Platz von Mann m in der Liste der Frau w. Offensichtlich sind die Werte dieser Hilfs-Arrays konstant und können zu Beginn aus den Werten von wmr und mwr bestimmt werden.

Der Prozess zur Bestimmung des Prädikats "stabil" läuft nun entsprechend der ursprünglichen Definition ab. Bekanntlich untersuchen wir die Möglichkeit der Heirat zwischen m und w, wobei w = wmr[m,r], d.h. w hat den Platz r in der Liste der Bevorzugungen von m. Als Optimisten nehmen wir zunächst an, dass die Stabilität erhalten bleibt, und versuchen dann, mögliche Quellen für Schwierigkeiten zu finden. Wo könnten sie verborgen sein? Es gibt zwei symmetrische Möglichkeiten:

1. Es könnte eine Frau pw geben, die von m seiner Frau w vorgezogen wird, und diese Frau selbst zieht m ihrem Mann vor.

2. Es könnte einen Mann pm geben, der von w ihrem Mann m vorgezogen wird, während er selbst w seiner Frau vorzieht.

Verfolgen wir Fall 1 und vergleichen die Plätze rwm[pw,m] und

rwm[pw,y[pw]] für alle Frauen, die m der Frau w vorzieht, d.h. für alle pw = wmr[m,i] mit i<r. Wir wissen, dass alle in Frage kommenden Frauen bereits verheiratet sind, da eine ledige schon vorher für m ausgesucht worden wäre. Der hier beschriebene Vorgang kann als eine einfache lineare Suche formuliert werden; dabei steht s für die Stabilität.

```
s := true; i := 1;                                          (3.45)
while (i<r) and "stabil" do
   begin pw := wmr[m,i];  i := i+1;
      if not single[pw] then s := rwm[pw,m] > rwm[pw,y[pw]]
   end
```

Verfolgen wir Fall 2, so müssen wir alle Kandidaten pm untersuchen, die w ihrem gegenwärtig angetrauten m vorzieht, d.h. in Frage kommen alle vorgezogenen Männer pm = wmr[w,i] mit i<rwm[w,m]. In Analogie zu Fall 1 sind Vergleiche zwischen den Plätzen rmw[pm,w] und rmw[pm,x[pm]] notwendig. Wir müssen dabei vorsichtig vorgehen und die Vergleiche mit x[pm] auslassen, wenn pm noch ledig ist. Dies erfordert einen Test pm<m, da bekanntlich alle Männer vor m bereits verheiratet sind.

Programm 3.6 enthält den vollständigen Algorithmus. Tabelle 3.3 gibt einen Satz von Eingabe-Werten für die Arrays wmr und mwr, und Tabelle 3.4 schliesslich enthält die neun berechneten stabilen Lösungen.

```
program marriage(input,output);
{Problem der stabilen Heirat}
const n = 8;
type man = 1..n; woman = 1..n; rank = 1..n;
var m: man; w: woman; r: rank;
   wmr: array [man, rank] of woman;
   mwr: array [woman, rank] of man;
   rmw: array [man, woman] of rank;
   rwm: array [woman, man] of rank;
   x:   array [man] of woman;
   y:   array [woman] of man;
   single: array [woman] of boolean;
```

```
procedure print;
   var m: man; rm, rw: integer;
begin rm := 0; rw := 0;
   for m := 1 to n do
   begin write(x[m]:4);
      rm := rm + rmw[m,x[m]]; rw := rw + rwm[x[m],m]
   end ;
   writeln(rm:8,rw:4);
end {print} ;

procedure try(m: man);
   var r: rank; w: woman;

   function stable: boolean;
      var pm: man; pw: woman;
          i, lim: rank;  s: boolean;
   begin s := true; i := 1;
      while (i<r)  and  s do
      begin pw := wmr[m,i]; i := i+1;
         if not single[pw] then s := rwm[pw,m] > rwm[pw,y[pw]]
      end ;
      i := 1; lim := rwm[w,m];
      while (i<lim) and s do
      begin pm := mwr[w,i]; i := i+1;
         if pm < m then s := rmw[pm,w] > rmw[pm,x[pm]]
      end ;
      stable := s
   end {test} ;

begin {try}
   for r := 1 to n do
   begin w := wmr[m,r];
      if single[w] then
         if stable then
         begin x[m] := w; y[w] := m; single[w] := false;
            if m < n then try(succ(m)) else print;
            single[w] := true
         end
   end
end {try} ;
```

```
begin
   for m := 1 to n do
      for r := 1 to n do
      begin read(wmr[m,r]); rmw[m,wmr[m,r]] := r
      end ;
   for w := 1 to n do
      for r := 1 to n do
      begin read(mwr[w,r]); rwm[w,mwr[w,r]] := r
      end ;
   for w := 1 to n do single[w] := true;
   try(1)
end .
```

Programm 3.6: Stabile Heirat

Reihenfolge		1	2	3	4	5	6	7	8
Mann 1	wählt Frau	7	2	6	5	1	3	8	4
2		4	3	2	6	8	1	7	5
3		3	2	4	1	8	5	7	6
4		3	8	4	2	5	6	7	1
5		8	3	4	5	6	1	7	2
6		8	7	5	2	4	3	1	6
7		2	4	6	3	1	7	5	8
8		6	1	4	2	7	5	3	8
Frau 1	wählt Mann	4	6	2	5	8	1	3	7
2		8	5	3	1	6	7	4	2
3		6	8	1	2	3	4	7	5
4		3	2	4	7	6	8	5	1
5		6	3	1	4	5	7	2	8
6		2	1	3	8	7	4	6	5
7		3	5	7	2	4	1	8	6
8		7	2	8	4	5	6	3	1

Tabelle 3.3

	x1	x2	x3	x4	x5	x6	x7	x8	rm	rw	c
Lösung 1	7	4	3	8	1	5	2	6	16	32	21
2	2	4	3	8	1	5	7	6	22	27	449
3	2	4	3	1	7	5	8	6	31	20	59
4	6	4	3	8	1	5	7	2	26	22	62
5	6	4	3	1	7	5	8	2	35	15	47
6	6	3	4	8	1	5	7	2	29	20	143
7	6	3	4	1	7	5	8	2	38	13	47
8	3	6	4	8	1	5	7	2	34	18	758
9	3	6	4	1	7	5	8	2	43	11	34

c = Anzahl der Tests auf Stabilität
Lösung 1 = optimale Lösung für die Männer
Lösung 2 = optimale Lösung für die Frauen

Tabelle 3.4

Dieser Algorithmus stützt sich wesentlich auf das Backtracking-Schema. Seine Effizienz hängt primär davon ab, wie weit das Schema zur Beschneidung des Lösungsbaumes entwickelt ist. Ein etwas schnellerer, aber komplexerer und undurchsichtigerer Algorithmus wurde von McVitie und Wilson vorgelegt ([21] und [22]), die ihn auch auf verschieden grosse Mengen (von Männern und Frauen) ausgedehnt haben.

Algorithmen von der Art der letzten beiden Beispiele, die alle möglichen Lösungen (unter gewissen Nebenbedingungen) generieren, werden oft verwendet, um eine oder mehrere, in gewisser Beziehung optimale Lösungen auszuwählen. Bei diesem Beispiel könnte man an der Lösung interessiert sein, die dem Wunsch der Männer - oder Frauen - oder aller Personen - im Mittel am besten gerecht wird.

Dazu enthält Tabelle 3.4 die Summe der Plätze aller Frauen in der Liste der Bevorzugungen ihrer Ehemänner und die Summen der Plätze aller Männer in der Liste ihrer Ehefrauen. Dies sind die Werte

$$rm = \sum_{m=1}^{n} rmw[m,x[m]], \quad rw = \sum_{m=1}^{n} rwm[x[m],m] \qquad (3.46)$$

Die Lösung mit dem kleinsten Wert rm heisst "für die Männer optimale stabile Lösung", die mit dem kleinsten rw ist die "für die Frauen optimale stabile Lösung". Durch die Art der gewählten Suchstrategie werden die für die Männer günstigen Lösungen zuerst generiert, die aus der Perspektive der Frauen günstigen Lösungen erscheinen am Ende. Der Algorithmus bevorzugt in diesem Sinn die männliche Bevölkerung. Dies ist (in der Abstraktion) durch systematisches Vertauschen der Rolle von Mann und Frau leicht zu ändern, nämlich durch Vertauschen von mwr mit wmr und rmw mit rwm.

Wir sehen davon ab, dieses Programm weiter auszubauen und behandeln die Suche nach einer optimalen Lösung im nächsten und letzten Beispiel eines Algorithmus mit Backtracking.

3.4.3. Das optimale Auswahlproblem

Das letzte Beispiel eines Backtracking-Algorithmus ist eine logische Erweiterung der beiden vorangehenden Beispiele, die dem allgemeinen Schema (3.35) entsprechen. Zunächst verwendeten wir das Prinzip des Backtracking zur Suche einer einzigen Lösung für ein gestelltes Problem. Dies wurde veranschaulicht durch den "Weg des Springers" und die "acht Damen". Dann stellten wir uns die Aufgabe, alle Lösungen eines gestellten Problems zu finden; dies waren die Beispiele der acht Damen und der stabilen Heirat. Nun wollen wir die optimale Lösung finden.

Dazu ist es notwendig, alle möglichen Lösungen zu generieren und sich - während der Generierung - zu merken, welche Lösung in gewisser Hinsicht optimal ist. Unter der Annahme, dass die Optimalität durch eine positivwertige Funktion f(s) definiert ist, leitet sich der Algorithmus von Schema (3.35) ab durch Ersetzen der Anweisung "drucke Lösung" durch die Anweisung

```
if f(solution) > f(optimum) then optimum := solution    (3.47)
```

Die Variable optimum enthält die beste bisher gefundene Lösung. Natürlich muss sie geeignet initialisiert werden; ausserdem ist es üblich, den Wert f(optimum) in einer anderen Variablen zu speichern, um so unnötige Wiederholungen seiner Berechnung zu

vermeiden.

Es folgt nun ein Beispiel für das allgemeine Problem des Findens einer optimalen Lösung zu einer gegebenen Aufgabe: Wir wählen das wichtige und gängige Problem des Findens einer optimalen Auswahl aus einer gegebenen Menge von Objekten unter gewissen Nebenbedingungen. Auswahlen, die annehmbare Lösungen darstellen, werden nach und nach durch Untersuchen der einzelnen Objekte der zugrunde liegenden Menge generiert. Eine Prozedur try soll die Untersuchung der Eignung eines Objektes beschreiben und wird so lange rekursiv aufgerufen (zur Untersuchung des nächsten Objektes), bis alle Objekte an der Reihe waren.

Die Betrachtung jedes Objektes (in den bisherigen Beispielen Kandidaten genannt) führt zu zwei möglichen Entscheidungen, entweder wird das untersuchte Objekt in die Auswahl eingeschlossen oder aus ihr ausgeschlossen. Dadurch wird die Verwendung der repeat- oder for-Anweisung unvorteilhaft, und die beiden Fälle werden explizit ausgeschrieben. Dies ist dargestellt in (3.48) unter der Annahme, dass die Objekte von 1 bis n durchnumeriert sind.

```
procedure try(i: integer);                                    (3.48)
begin
(1) if "Einschluss möglich" then
    begin "i-tes Objekt einschliessen";
       if i < n then try(i+1) else "prüfe Optimalität";
       "eliminiere i-tes Objekt"
    end;
(2) if "Ausschluss möglich" then
       if i < n then try(i+1) else "prüfe Optimalität"
end
```

Hieraus ist ersichtlich, dass es 2**n mögliche Mengen gibt; sicher müssen geeignete Kriterien zur Annahme verwendet werden, um die Zahl der untersuchten Kandidaten drastisch zu reduzieren. Diesen Prozess wollen wir an einem konkreten Beispiel für das Auswahlproblem beleuchten: Sei jedes der n Objekte a[1], ... , a[n] durch sein Gewicht und seinen Wert v beschrieben. Die Menge mit der grössten Summe der Werte ihrer Komponenten sei optimal, und die Summe der Gewichte sei beschränkt. Alle kofferpackenden Reisenden

kennen dieses Problem, nämlich die Auswahl von n Gegenständen, deren gesamter Wert optimal sein soll und deren Gesamtgewicht eine vorgeschriebene Grenze nicht überschreiten darf.

Wir sind jetzt in der Lage, über die Darstellung der gegebenen Tatsachen in Form von Daten zu entscheiden. Die Wahl (3.49) lässt sich leicht aus obigen Ueberlegungen ableiten.

```
type index = 1..n;                                    (3.49)
     object = record w,v: integer end;
var a: array [index] of object;
    limw, totv, maxv: integer;
    s, opts: set of index
```

Die Variablen limw und totv bezeichnen die Gewichtsgrenze und den Gesamtwert aller n Objekte. Diese beiden Werte sind während des ganzen Auswahlprozesses konstant. s stellt die momentane Auswahl der Objekte dar, wobei jedes Objekt durch seinen Namen (Index) dargestellt ist. opts ist die beste bisher gefundene Wahl, und maxv ist ihr Wert.

Welches sind nun die Kriterien für die Eignung eines Objektes für die vorliegende Wahl? Beim Betrachten von Einschluss ist ein Objekt annehmbar, wenn es in die Gewichtsgrenze passt. Betrachten wir dagegen den Ausschluss, dann ist das Kriterium für die Eignung, d.h. für den weiteren Aufbau der vorliegenden Auswahl, ob der nach dem Ausschluss erreichbare Gesamtwert nicht kleiner ist als der Wert des bisher gefundenen Optimums. Ist er nämlich kleiner, führt die Fortsetzung der Suche wohl zu einer Lösung, aber nicht zur optimalen Lösung. Jedes weitere Suchen auf diesem Weg ist also umsonst. Aus diesen beiden Bedingungen bestimmen wir die für jeden Schritt des Auswahlprozesses wesentlichen Werte:

1. Das Gesamtgewicht tw der bisherigen Auswahl s.
2. Den mit der gegenwärtigen Auswahl s immer noch erreichbaren Wert av.

Diese beiden Grössen sind geeignet dargestellt als Parameter der Prozedur try.

Die Bedingung "Einschluss möglich" in (3.48) kann nun formuliert werden als

```
tw + a[i].w <= limw                                          (3.50)
```

und der anschliessende Test auf Optimalität lautet

```
if av > maxv then                                            (3.51)
   begin {zeichne neues Optimum auf}
      opts := s;  maxv := av
   end
```

Die letzte Zuweisung stützt sich auf die Ueberlegung, dass der erreichbare Wert gleich dem erreichten Wert ist, sobald alle n Objekte behandelt wurden.

Die Bedingung "Ausschluss möglich" in (3.48) wird ausgedrückt durch

```
av - a[i].v > maxv                                           (3.52)
```

Da der Wert av-a[i].v > maxv später nochmals verwendet wird, weisen wir ihm den Namen av1 zu und vermeiden so eine erneute Berechnung.

Das ganze Programm ergibt sich nun aus (3.48) bis (3.52) unter Hinzufügung geeigneter Initialisierungs-Anweisungen für die globalen Variablen. Beachtenswert ist die Leichtigkeit, mit der Einschluss oder Ausschluss aus der Menge s durch die Verwendung der Strukturart Menge (set) ausgedrückt werden kann. Die Resultate der Ausführung von Programm 3.7 mit zulässigen Gewichten zwischen 10 und 120 sind in Tabelle 3.5 zusammengefasst.

```
program selection(input,output);
{Suche eine optimale Auswahl von Objekten unter
Nebenbedingungen}
const n = 10;
type index = 1..n;
   object = record v,w: integer end ;
var i: index;
   a: array [index] of object;
   limw, totv, maxv: integer;
```

```
   w1, w2, w3: integer;
   s, opts: set of index;
   z: array [boolean] of char;

procedure try(i: index; tw,av: integer);
   var av1: integer;
begin {Versuche den Einschluss von Objekt i}
   if tw + a[i].w <= limw then
   begin s := s + [i];
      if i < n then try(i+1, tw+a[i].w, av) else
         if av > maxv then
         begin maxv := av; opts := s
         end :
      s := s - [i]
   end :
   {Versuche nun ohne Objekt i}  av1 := av - a[i].v;
   if av1 > maxv then
   begin if i < n then try(i+1, tw, av1) else
            begin maxv := av1; opts := s
            end
   end
end {try} ;

begin totv := 0;
   for i := 1 to n do
      with a[i] do
      begin read(w,v); totv := totv + v
      end :
   read(w1,w2,w3);
   z[true] := '*'; z[false] := ' ';
   write(' Gewicht  ');
   for i := 1 to n do write(a[i].w:4);
   writeln; write(' Wert     ');
   for i := 1 to n do write(a[i].v:4);
   writeln;
   repeat limw := w1; maxv := 0; s := []; opts := [];
      try(1,0,totv);
      write(limw);
```

```
      for i := 1 to n do write('   ', z[i in opts]);
      writeln; w1 := w1 + w2
   until w1 > w3
end .
```

Programm 3.7: Optimale Auswahl

Gewicht	10	11	12	13	14	15	16	17	18	19
Wert	18	20	17	19	25	21	27	23	25	24
10	*									
20							*			
30					*		*			
40	*				*		*			
50	*	*		*			*			
60	*	*	*	*	*					
70	*	*			*		*		*	
80	*	*	*		*		*	*		
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*

Tabelle 3.5

Dieses Backtracking-Schema mit einem Faktor, der das Wachsen des möglichen Suchbaumes beschränkt, ist auch bekannt als branch-and-bound-Algorithmus.

Uebungen

3.1. (Türme von Hanoi) Gegeben seien drei Stäbe und n Scheiben verschiedener Grössen. Die Scheiben können auf die Stäbe gesteckt werden und bilden so "Türme". Die n Scheiben sitzen zu Beginn auf dem Stab A, nach abnehmender Grösse geordnet, wie es in Fig. 3.10 für n = 3 gezeigt ist.

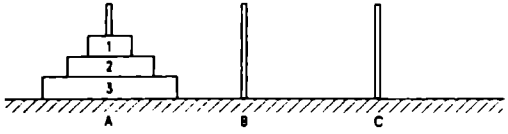


Fig. 3.10: Die "Türme von Hanoi"

Die Aufgabe besteht nun darin, die n Scheiben vom Stab A nach Stab C derart zu bringen, dass ihre ursprüngliche Ordnung erhalten bleibt. Dabei gelten folgende Einschränkungen:

1. Bei jedem Schritt wird genau eine Scheibe von einem Stab zum anderen bewegt.
2. Eine Scheibe darf nie auf einer kleineren liegen.
3. Stab B darf als Zwischenspeicher verwendet werden.

Man bestimme einen Algorithmus, der diese Aufgabe löst. Man beachte, dass der Turm sehr wohl so betrachtet werden kann, als setze er sich zusammen aus einer einzelnen Scheibe und aus einem darunterliegenden Turm, der aus den übrigen Scheiben besteht. Dieser Algorithmus ist durch ein rekursives Programm zu beschreiben.

3.2. Man schreibe eine Prozedur, die alle n! Permutationen von n Elementen a[1], ... , a[n] in situ erzeugt, d.h. ohne die Hilfe eines weiteren Array. Nach der Generierung jeder Permutation soll eine parameterisierte Prozedur Q aufgerufen werden, die z.B. die generierte Permutation ausdruckt.

Hinweis: Man betrachte die Aufgabe der Generierung aller Permutationen der Elemente a[1], ... , a[m] als bestehend aus den m Teilaufgaben der Generierung aller Permutationen von a[1], ... , a[m-1], gefolgt von a[m], wobei bei der i-ten Teilaufgabe die beiden Elemente a[i] und a[m] zu Beginn ausgetauscht wurden.

3.3. Man leite das Rekursionsschema von Fig. 3.11 ab, die eine Ueberlagerung der vier Kurven W(1), W(2), W(3) und W(4) ist. Die Struktur ähnelt der der Sierpinski-Kurven (3.21) und (3.22). Aus dem Rekursionsmuster leite man ein rekursives Programm ab, das diese Kurven zeichnet.

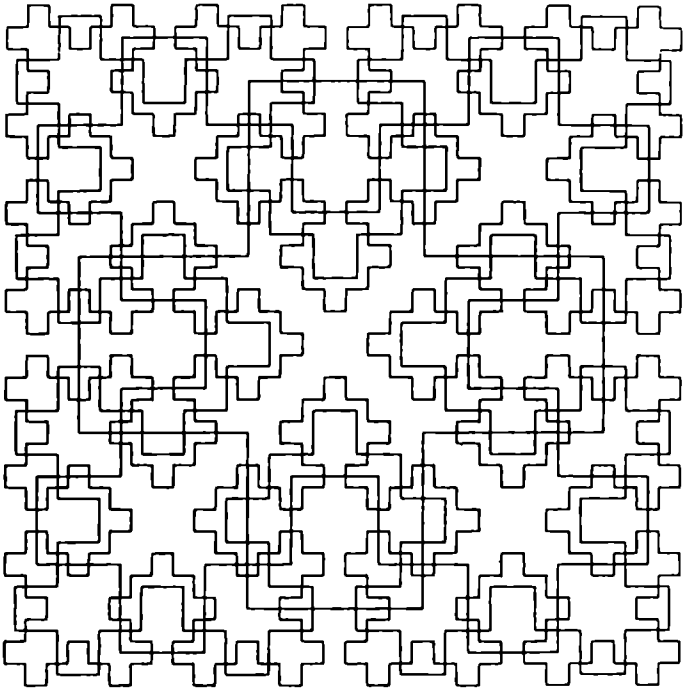


Fig. 3.11: Die Kurven W(1) ... W(4)

3.4. Nur 12 der 92 Lösungen, die durch das Programm 3.5 der acht Damen berechnet werden, sind wirklich voneinander verschieden. Die

anderen kann man durch Spiegelungen an den Achsen oder um den Mittelpunkt herleiten. Man entwerfe ein Programm zur Bestimmung der 12 Hauptlösungen. Man beachte, dass beispielsweise das Suchen in Kolonne 1 auf die Positionen 1 bis 4 beschränkt werden kann.

3.5. Aendere das Programm der stabilen Heirat, so dass es die beste Lösung (für Männer oder Frauen) bestimmt. Es wird dann zu einem "branch-and-bound" Programm vom Typ des Programms 3.7.

3.6. Eine gewisse Eisenbahngesellschaft bedient n Stationen S1,..., Sn. Sie möchte den Informationsdienst für ihre Kunden durch rechnergesteuerte Datenstationen verbessern. Ein Kunde tippt seine Abfahrtsstation SA und sein Ziel SZ ein und soll (sofort) den Fahrplan der Zugverbindungen mit der kürzesten Reisezeit erhalten.

Man entwerfe ein Programm zur Berechnung der gewünschten Information. Man gehe davon aus, dass der Fahrplan (die Datenbank) in geeigneter Struktur vorliegt und die Abfahrtszeiten (= Ankunftszeiten) aller möglichen Züge enthält. Selbstverständlich bestehen nicht zwischen allen Stationen direkte Verbindungen (vgl. auch Aufgabe 1.8).

3.7. Die Ackermann'sche Funktion A ist für alle nicht negativen ganzzahligen Argumente m und n folgendermassen definiert:

$$
\begin{aligned}
A(0,n) &= n+1 \\
A(m,0) &= A(m-1,1) && (m > 0) \\
A(m,n) &= A(m-1,A(m,n-1)) && (m,n > 0)
\end{aligned}
$$

Man entwerfe ein Programm zur Berechnung von A(m,n) ohne Verwendung von Rekursion.

Als Leitfaden verwende man Programm 2.11, die nicht rekursive Version des Quicksort. Man stelle für die Umformung eines rekursiven Programms in ein iteratives einen Satz allgemeiner Regeln auf.

4. Dynamische Informationsstrukturen

4.1. Rekursive Datentypen

In Kapitel 2 wurden die Array-, Record- und Mengen-Strukturen als grundlegende Datenstrukturen eingeführt. Sie heissen grundlegend, weil sie die Bausteine für komplexere Strukturen sind und in der Praxis sehr häufig vorkommen. Der Sinn der Definition eines Datentyps und einer späteren Spezifikation einer Variablen von diesem Typ ist die Festlegung des durch diese Variable angenommenen Wertebereichs und damit auch ihres Speicherschemas. So vereinbarte Variable heissen daher statisch. Es gibt aber viele Probleme, die weit kompliziertere Datenstrukturen erfordern. Wesentlich für diese Probleme ist die Aenderung ihrer Strukturen während der Ausführung. Sie werden deshalb dynamische Strukturen genannt. Natürlich sind die Komponenten solcher Strukturen - auf irgendeiner tieferen Stufe - statisch, d.h. von einem Grundtyp. Dieses Kapitel ist der Konstruktion, Analyse und Verwaltung dynamischer Informationsstrukturen gewidmet.

Es ist bemerkenswert, welch grosse Aehnlichkeit zwischen den Methoden zur Strukturierung von Algorithmen und denen zur Strukturierung von Daten besteht. Wie bei jeder Aehnlichkeit gibt es einige Unterschiede (sonst wäre es eine Identität), aber ein Vergleich der Methoden der Strukturierung für Programme und Daten ist dennoch aufschlussreich.

Die elementare unstrukturierte Anweisung ist die Zuweisung. Das entsprechende Mitglied in der Familie der Datenstrukturen ist der skalare unstrukturierte Typ. Diese beiden sind die unteilbaren Bausteine für zusammengesetzte Anweisungen und Datentypen. Die einfachsten Strukturen, die man durch Aufzählung oder Aneinanderreihen erhält, sind die zusammengesetzte Anweisung und die Record-Struktur. Beide bestehen aus einer endlichen (gewöhnlich kleinen) Zahl von explizit aufgeführten Komponenten, die untereinander selbst alle verschieden sein können. Wenn alle Komponenten identisch sind, brauchen sie nicht einzeln ausgeschrieben zu werden:

Wir verwenden die for-Anweisung und die Array-Struktur, um die Wiederholung mit einem bekannten festen Faktor anzugeben. Eine Wahl zwischen zwei oder mehr Varianten wird ausgedrückt durch die bedingte oder die case-Anweisung, bzw. durch die variante Record-Struktur. Die Wiederholung um einen anfänglich unbekannten (und möglicherweise unbegrenzten) Faktor wird ausgedrückt durch die while- oder die repeat-Anweisung. Die entsprechende Datenstruktur ist die Sequenz (file), die einfachste Art der Konstruktion von Typen mit unendlicher Kardinalität.

Es erhebt sich die Frage nach der Existenz einer Datenstruktur, die auf ähnliche Weise der Prozedur-Anweisung entspricht. Natürlich ist in diesem Zusammenhang die interessanteste und neuartige Eigenschaft der Prozeduren die Rekursion. Werte solch rekursiver Datentypen könnten eine oder mehrere Komponenten enthalten, die zum gleichen Typ gehören wie sie selbst, in Analogie zu einer Prozedur, die einen oder mehrere Aufrufe von sich selbst enthält. Wie Prozeduren könnten solche Typen-Definitionen direkt oder indirekt rekursiv sein.

Ein einfaches Beispiel eines Objekts, das sich sehr gut für die Darstellung durch einen rekursiv definierten Typ eignet, ist der arithmetische Ausdruck, wie er in Programmiersprachen vorkommt. Rekursion wird verwendet, um die Möglichkeit der Verschachtelung wiederzuspiegeln, d.h. um geklammerte Teilausdrücke als Operanden in Ausdrücken zu verwenden. Es sei folglich ein Ausdruck auf folgende informelle Art definiert:

> Ein Ausdruck besteht aus einem Term, gefolgt von einem Operator, gefolgt von einem Term. (Die beiden Terme bilden die Operanden für den Operator.) Ein Term ist entweder eine Variable - dargestellt durch einen Namen - oder ein eingeklammerter Ausdruck.

Ein Datentyp, dessen Werte solche Ausdrücke darstellen, kann leicht durch bereits vorhandene Hilfsmittel unter Hinzunahme der Rekursion beschrieben werden.

```
type expression = record op: operator;                    (4.1)
                        opd1, opd2: term
                  end;
```

```
term = record
          if t then (id: alfa)
               else (subex: expression)
       end
```

Folglich besteht jede Variable vom Typ Term aus zwei Komponenten, nämlich dem Typ-Diskriminator t und dem Feld id für t gleich true oder aus dem Feld subex im anderen Fall. Betrachten wir nun z.B. die folgenden vier Ausdrücke (4.2).

1. x+y (4.2)
2. x-(y*z)
3. (x+y)*(z-w)
4. (x/(y+z))*w

Diese Ausdrücke können durch die Schemata in Fig. 4.1 veranschaulicht werden, die die verschachtelte rekursive Struktur aufzeigen und die die Anordnung oder Abbildung dieser Ausdrücke in einem Speicher bestimmen.

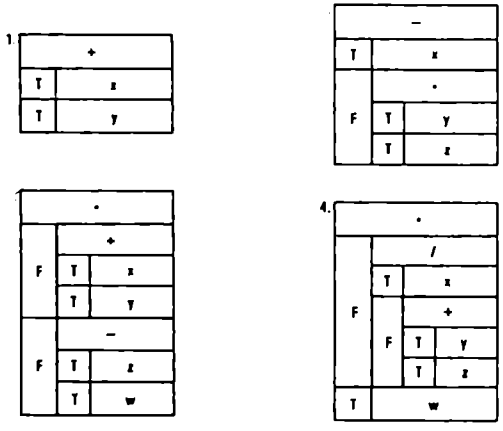


Fig. 4.1: Speicherbild rekursiver Record-Strukturen

Ein zweites Beispiel einer rekursiven Informationsstruktur ist der Familienstammbaum: Der Stammbaum sei definiert durch eine Person (bzw. ihren Namen) und die beiden Stammbäume der Eltern. Diese Definition führt unausweichlich zu einer unendlichen Struktur.

Echte Stammbäume sind begrenzt, da auf irgendeiner Stufe Information über die Vorfahren fehlt. Dieser Tatsache kann Rechnung getragen werden, wenn man eine variante Struktur verwendet, wie sie in (4.3) gezeigt ist.

```
type Stammbaum = record                                    (4.3)
                   if known then
                      (name: alfa;
                       father, mother: Stammbaum)
                 and
```

(Man beachte, dass jede Variable vom Typ Stammbaum mindestens eine Komponente hat, nämlich das Diskriminatorfeld known. Ist sein Wert gleich true, dann gibt es drei weitere Felder, andernfalls keines.) Der spezielle Wert, der durch den (rekursiven) Record-Konstruktor

```
x = (T,Ted,(T,Fred,(T,Adam,(F),(F)),(F)),
                (T,Mary,(F),(T,Eva,(F),(F)))
```

gegeben ist, ist in Fig. 4.2 auf eine Art dargestellt, die ein mögliches Speicherschema andeutet. (Da nur ein einziger Record-Typ beteiligt ist, haben wir den Typbezeichner Stammbaum vor jedem Konstruktor weggelassen.)

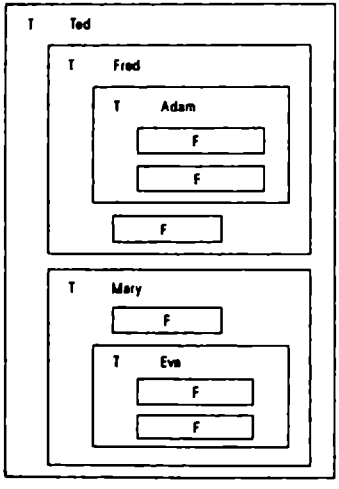


Fig. 4.2: Struktur eines Stammbaumes

Die wichtige Rolle der Variantenbildung wird dabei deutlich; sie ist das einzige Mittel zur Begrenzung einer rekursiven Datenstruktur und ist deshalb unerläßlicher Teil jeder rekursiven Definition. Die Analogie zwischen Konzepten zur Programm- und Datenstrukturierung wird in diesem Fall besonders offensichtlich. Eine bedingte Anweisung muss notwendigerweise Teil jeder rekursiven Prozedur sein, damit die Ausführung der Prozedur terminiert. Termination der Ausführung entspricht offensichtlich endlicher Kardinalität.

4.2. Zeiger oder Referenzen

Die wesentliche Eigenschaft rekursiver Strukturen, die sie deutlich von fundamentalen Strukturen (Arrays, Records und Mengen) unterscheidet, ist ihre Fähigkeit, ihre Grösse zu verändern. Daher ist es unmöglich, einer rekursiv definierten Struktur einen festen Speicherbereich zuzuweisen, und folglich kann ein Compiler den Komponenten solcher Variablen keine spezifischen Adressen zuordnen. Die weitverbreitete Technik zur Lösung dieses Problems verwendet dynamische Speicherzuordnung, d.h. Zuweisung von Speicher zu einzelnen Komponenten erst dann, wenn sie während der Programmausführung entstehen, und nicht schon während der Uebersetzung. Der Compiler weist dann für die Aufnahme der Adressen der dynamisch zugewiesenen Komponenten einen festen Speicherplatz zu und nicht für die Aufnahme der Komponente selbst. So würde z.B. der in Fig. 4.2 dargestellte Stammbaum durch einzelne - möglicherweise nicht zusammenhängende - Records dargestellt werden, und zwar durch einen Record für jede Person. Diese Personen sind dann verbunden durch ihre Adressen, die den jeweiligen Feldern father und mother zugewiesen werden. Graphisch läßt sich diese Situation am besten darstellen durch die Verwendung von Pfeilen oder Zeigern (siehe Fig. 4.3).

Es muss betont werden, dass die Verwendung von Zeigern zur Implementation rekursiver Strukturen nur ein Hilfsmittel ist. Dem Programmierer muss deren Existenz nicht bewusst sein. Sind die Zeiger dem Programmierer unzugänglich, so kann Speicherplatz auto-

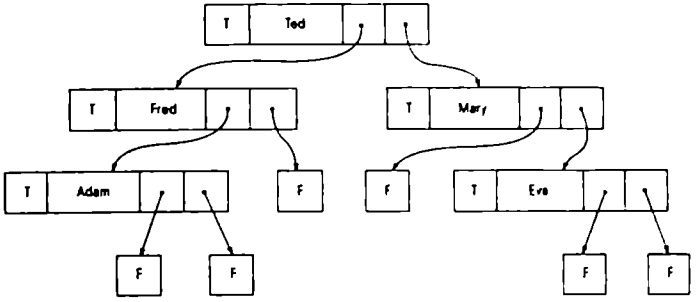


Fig. 4.3: Durch Zeiger verkettete Struktur

matisch bei der ersten Referenz einer neuen Komponente zugewiesen werden. Sind dagegen Zeiger explizit zugänglich, so können allgemeinere Datenstrukturen konstruiert werden, als dies durch rein rekursive Definition möglich wäre. Insbesondere ist es dann möglich, "unendliche" oder zirkuläre Strukturen zu definieren und zu erreichen, dass gewisse Strukturen gemeinsam verwendet werden. In neueren Programmiersprachen ist es daher üblich, die Handhabung von Referenzen zu Daten zusätzlich zu den Daten selbst explizit verfügbar zu machen. Dies bedingt eine klare notationelle Unterscheidung zwischen Daten und Referenzen zu Daten, und zusätzliche Datentypen sind einzuführen, deren Werte Zeiger (Referenzen) zu anderen Daten sind. Zu diesem Zweck verwenden wir die folgende Notation:

$$\underline{\text{type}}\ Tp = \uparrow T \qquad (4.4)$$

Die Typ-Vereinbarung (4.4) besagt, dass die Werte vom Typ Tp Zeiger zu Daten vom Typ T sind. So kann der Pfeil in (4.4) in Worten ausgedrückt werden als "Zeiger zu". Es ist von grundlegender Bedeutung, dass der Typ der Elemente, auf die gezeigt wird, aus der Vereinbarung des Zeigertyps ersichtlich ist. Wir sagen, Tp ist an T gebunden. Diese Bindung unterscheidet Zeiger in höheren Programmiersprachen von Adressen im Assembler-Code; sie ist zur Erhöhung der Sicherheit beim Programmieren durch die Redundanz der zugrunde liegenden Notation äusserst wichtig.

Werte von Zeigertypen werden generiert, wenn ein neues Datenelement

dynamisch zugewiesen wird. Wir wollen an der Konvention festhalten, dass solch ein Ereignis jedesmal explizit angegeben wird. Dies geschieht im Gegensatz zu der Situation, in der jede erste Erwähnung eines Elementes automatisch die Zuweisung vollzieht (oder vollziehen soll). Zu diesem Zweck führen wir die interne Standard-Prozedur new ein. Sei p eine Zeiger-Variable vom Typ Tp, dann weist die Anweisung

```
new(p)                                                    (4.5)
```

effektiv eine Variable vom Typ T zu, generiert einen Zeiger vom Typ Tp, der auf diese neue Variable zeigt, und weist diesen Zeiger der Variablen p zu (siehe Fig. 4.4).

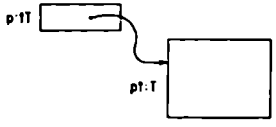


Fig. 4.4: Dynamische Speicherzuweisung an p↑

Der Zeigerwert selbst kann nun angesprochen werden als p (d.h. als Wert der Zeiger-Variablen p). Dagegen wird die durch p referenzierte Variable mit p↑ bezeichnet. Dies ist die dynamisch zugewiesene Variable vom Typ T.

Es wurde oben erwähnt, dass eine variante Komponente wesentlicher Bestandteil jedes rekursiven Typs ist, um endliche Kardinalität zu sichern. Das Beispiel des Familienstammbaumes ist ein Muster, das eine äusserst häufig auftretende Konstellation zeigt (vgl. (4.3)), nämlich den Fall, in dem der Typ-Diskriminator zweiwertig ist (Boolean) und sein Wert, wenn er gleich false ist, das Fehlen weiterer Komponenten angibt. Dies wird ausgedrückt durch das Vereinbarungsschema (4.6)

```
type T = record if p then S(T) and                        (4.6)
```

S(T) bezeichnet eine Folge von Feld-Definitionen, die ein oder mehrere Felder vom Typ T enthalten und somit für Rekursivität sorgen. Alle Strukturen eines Typs nach Muster (4.6) stellen eine Baum-

(oder Listen-) Struktur dar, ähnlich der von Fig. 4.3. Insbesondere enthalten sie Zeiger zu Datenkomponenten, die nur ein Diskriminatorfeld umfassen, d.h. keine weitere wesentliche Information enthalten. Die Technik der Implementation mit Zeigern ermöglicht eine einfache Art des Sparens von Speicherplatz, indem man die Diskriminator-Information in den Wert des Zeigers selbst aufnimmt. Die übliche Lösung ist die Erweiterung des Wertebereichs des Typs Tp durch einen einzigen Wert, der zu überhaupt keinem Element zeigt. Wir bezeichnen diesen Wert durch das Spezial-Symbol nil und postulieren, dass nil automatisch zum Wertebereich aller Zeigertypen gehört. Diese Erweiterung des Bereichs von Zeigerwerten macht deutlich, warum endliche Strukturen ohne explizites Vorhandensein von Varianten (Bedingungen) in ihrer (rekursiven) Vereinbarung generiert werden können.

Die neue Formulierung der in (4.1) und (4.3) vereinbarten Typen - gestützt auf die explizite Verwendung von Zeigern - wird in (4.7) bzw. (4.8) gegeben. Zu beachten ist im zweiten Fall - der ursprünglich dem Schema (4.6) entsprach - dass die variante Record-Komponente überflüssig wurde, da p↑.known = false nun ausgedrückt wird durch p = nil. Die Umbenennung des Typs Stammbaum in person spiegelt die Veränderung des Standpunkts wieder, der durch die Einführung expliziter Zeigerwerte ins Spiel gebracht wurde. Anstatt die gegebene Struktur zuerst in ihrer Gesamtheit zu betrachten und dann ihre Unterstruktur und ihre Komponenten zu untersuchen, wird die Aufmerksamkeit jetzt zuerst auf die Komponenten gerichtet, und ihre gegenseitige Beziehung (dargestellt durch Zeiger) ergibt sich nicht aus einer festen Vereinbarung.

```
type expression = record op: operator;                    (4.7)
                          opd1, opd2: ↑term
                  end;
     term = record
                if t then (id: alfa)
                     else (sub: ↑expression)
            end

type person = record name: alfa;                          (4.8)
                     father, mother: ↑person
              end
```

Die Datenstruktur, die den Stammbaum in Fig. 4.2 und 4.3 zeigt, wird nochmals dargestellt in Fig. 4.5, in der Zeiger zu unbekannten Personen durch nil angegeben werden. Die resultierende Verbesserung der Speicherbelegung ist offensichtlich.

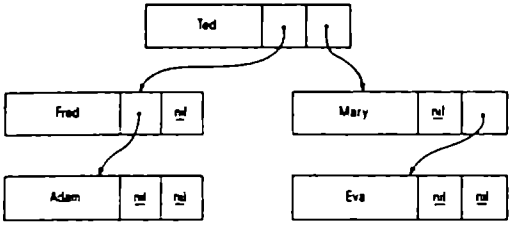


Fig. 4.5: Struktur mit nil-Zeigern

Fred und Mary seien - wieder mit Bezug auf Fig. 4.5 - Geschwister, d.h. sie haben denselben Vater und dieselbe Mutter. Diese Situation lässt sich leicht durch Ersetzen der beiden nil-Werte in den entsprechenden Feldern der beiden Records ausdrücken. Eine Implementation, die das Konzept der Zeiger versteckt oder eine andere Technik der Speicherbehandlung verwendet, würde den Programmierer zwingen, die Records von Adam und Eva je zweimal darzustellen. Obwohl es beim Zugriff zu ihren Daten für die Inspektion nichts ausmacht, ob der Vater (und die Mutter) der beiden durch zwei oder einen Record dargestellt ist, so wird der Unterschied doch wesentlich, wenn selektives Aendern erlaubt ist. Die Behandlung von Zeigern als explizite Datenelemente anstelle versteckter Implementationshilfen erlaubt dem Programmierer, klar zu sagen, wo Speicher gemeinsam benützt werden soll.

Eine weitere Konsequenz expliziter Zeiger ist die Möglichkeit zur Definition und Behandlung zyklischer Datenstrukturen. Diese zusätzliche Flexibilität gibt natürlich nicht nur mehr Möglichkeiten, sondern erfordert auch erhöhte Sorgfalt vom Programmierer, da die Behandlung zyklischer Datenstrukturen leicht zu nicht terminierenden Prozessen führen kann.

Dieses Phänomen, dass Vielseitigkeit und Flexibilität eng mit der Gefahr des Missbrauchs verknüpft sind, ist bei der Programmierung bekannt und erinnert speziell an die goto-Anweisung. In der Tat

kann, wenn man die Analogie zwischen Programm und Datenstrukturen erweitert, die rein rekursive Datenstruktur auf die entsprechende Stufe mit der Prozedur gestellt werden, während die Einführung von Zeigern mit der Verwendung der goto-Anweisung vergleichbar ist. Denn so wie die goto-Anweisung die Konstruktion irgendeines Programm-Musters erlaubt (inklusive Schleifen), so erlauben Zeiger die Erstellung jeder Art von Datenstrukturen (inklusive Zyklen). Die parallele Entwicklung entsprechender Programm- und Datenstrukturen ist in Tabelle 4.1 zusammengestellt.

Konstruktions-Muster	Programm-Anweisung	Daten-Typ
atomares Element	Zuweisung	skalarer Typ
Aufzählung	zusammengesetzte Anweisung	Record-Typ
Wiederholung um bekannte Anzahl	for-Anweisung	Array-Typ
Auswahl	bedingte Anweisung	varianter Record, Typen-Vereinigung
Wiederholung um unbekannte Anzahl	while- oder repeat-Anweisung	Sequenz- oder File-Typ
Rekursion	Prozedur-Anweisung	rekursiver Datentyp
allgemeiner Graph	goto- Anweisung	durch Zeiger verkettete Struktur

Tabelle 4.1

In Kapitel 3 wurde gezeigt, dass die Iteration ein Spezialfall der Rekursion ist und dass der Aufruf einer rekursiven Prozedur P entsprechend dem Schema (4.9)

```
procedure P;                                              (4.9)
begin
   if B then begin P0; P end
end
```

mit P0 als Anweisung, die P nicht aufruft, äquivalent ist und ersetzt werden kann durch die iterative Anweisung

```
while B do P0 .
```

Die in Tabelle 4.1 aufgeführten Analogien zeigen, dass eine ähnliche Beziehung zwischen rekursiven Datentypen und der Sequenz besteht. Tatsächlich ist ein nach Schema (4.10) vereinbarter rekursiver Typ

```
type T = record                                    (4.10)
           if B then (t0: T0; t: T)
         and
```

mit einem Typ T0, der T nicht enthält, äquivalent und kann ersetzt werden durch den sequentiellen Datentyp

```
file of T0 .
```

Dies zeigt, dass Rekursion durch Iteration in Programm und Datendefinition dann (und nur dann) ersetzt werden kann, wenn die Prozedur oder der Typ-Name nur einmal rekursiv am Ende (oder zu Beginn) seiner Definition vorkommt.

Der Rest dieses Kapitels ist der Erstellung und Behandlung von Datenstrukturen gewidmet, deren Komponenten durch explizite Zeiger verknüpft sind. Strukturen mit speziell einfachen Mustern werden besonders hervorgehoben; Rezepte für die Behandlung komplexerer Strukturen können aus den Regeln zur Behandlung einfacherer Gliederungen abgeleitet werden. Dies sind lineare Listen oder verkettete Sequenzen - der einfachste Fall - und Bäume. Wenn wir uns hauptsächlich mit diesen "Bausteinen" der Datenstrukturierung beschäftigen, so folgt daraus nicht, dass in Wirklichkeit nicht kompliziertere Strukturen vorkommen. Tatsächlich ist die folgende Geschichte, die in einer Zeitung im Juli 1922 in Zürich erschien, ein Beweis dafür, dass Unregelmässigkeiten sogar in solchen Fällen auftreten können, die gewöhnlich als Beispiele für reguläre Strukturen dienen, wie z.B. (Familien-)Bäume. Ein Mann beschrieb dort die Tragik seines Lebens mit folgenden Worten:

> Ich verheiratete mich mit einer Witwe, die eine erwachsene Tochter hatte. Mein Vater, der uns oft besuchte, verliebte sich in meine Stieftochter und heiratete sie; dadurch wurde mein Vater mein Schwiegersohn und meine Stieftochter meine Mutter. Einige Zeit darauf schenkte mir meine Frau einen Sohn, der der Schwager meines Vaters und mein Onkel wurde. Die Frau meines

Vaters, meine Stieftochter, bekam auch einen Sohn. Dadurch erhielt ich einen Bruder und gleichzeitig einen Enkel. Meine Frau ist meine Grossmutter, da sie ja die Mutter meiner Mutter ist. Ich bin also der Mann meiner Frau und gleichzeitig der Stiefenkel meiner Frau; mit anderen Worten, ich bin mein eigener Grossvater.

4.3. Lineare Listen

4.3.1. Grundoperationen

Verbindungen zwischen einer Menge von Elementen werden am einfachsten hergestellt, indem man sie in einer einzigen Liste oder Schlange aufreiht. Denn in diesem Fall ist für jedes Element nur eine Verbindung nötig, um auf den Nachfolger zu verweisen.

Der Typ T sei gemäss (4.11) definiert. Jede Variable von diesem Typ besteht aus drei Komponenten, nämlich einem identifizierenden Schlüssel (key), dem Zeiger zu ihrem Nachfolger und etwaiger weiterer Information, die in (4.11) ausgelassen wird.

```
type T = record key: integer;                    (4.11)
                next: ↑T;
                ...
         end
var p: ↑T;
```

Eine Liste von Elementen vom Typ T mit einem Zeiger zu ihrer ersten Komponente in der Variablen p ist in Fig. 4.6 dargestellt.

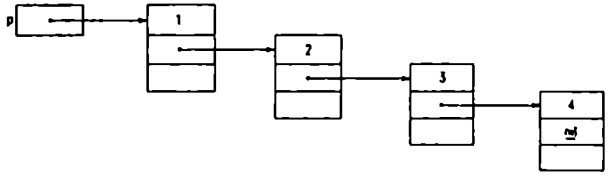


Fig. 4.6: Beispiel einer Liste

Vielleicht die einfachste Operation, die auf einer wie in Fig. 4.6 gezeigten Liste ausgeführt werden kann, ist das Anfügen eines Elementes an ihrem Kopf. Zuerst wird ein Element vom Typ T zugeordnet und der zugehörige Zeiger einer Hilfs-Variablen, nennen wir sie q, zugewiesen. Dann schliesst ein einfaches Umspeichern der Zeiger die Operation ab, die in (4.12) programmiert ist.

```
new(q); q↑.next := p; p := q                                    (4.12)
```

Man beachte, dass die Reihenfolge dieser drei Anweisungen wesentlich ist.

Die Operation des Anfügens eines Elementes am Kopf einer Liste legt sofort nahe, wie eine solche Liste generiert werden kann: Ausgehend von der leeren Liste wird ständig ein Kopfelement angefügt. Dieser Prozess der Listengenerierung ist in (4.13) formuliert; hier ist die Zahl der zu verbindenden Elemente n.

```
p := nil; {beginne mit leerer Liste}                            (4.13)
while n > 0 do
   begin new(q); q↑.next := p; p := q;
         q↑.key := n; n := n-1
   end
```

Dies ist die einfachste Art, eine Liste zu bilden, aber die sich ergebende Reihenfolge der Elemente ist umgekehrt zur Reihenfolge ihrer "Ankunft". In einigen Anwendungen ist dies unerwünscht; folglich müssen neue Elemente am Ende der Liste angehängt werden. Obwohl das Ende durch ein Durchlaufen der Liste leicht bestimmt werden kann, benötigt dieser naive Versuch einen Aufwand, der sehr wohl vermieden werden kann durch die Verwendung eines zweiten Zeigers, nennen wir ihn q, der immer auf das letzte Element weist. Diese Methode ist z.B. in Programm 4.4 angewendet, das eine Cross-Reference-Liste zu einem gegebenen Text erzeugt. Als nachteilig erweist sich die Behandlung des ersten einzufügenden Elementes, da diese von allen weiteren verschieden ist.

Die ausdrückliche Verfügbarkeit der Zeiger macht gewisse Operationen sehr einfach, die sonst recht umständlich wären; unter den elementaren Listenoperationen befinden sich auch das Einschieben

und Entfernen von Elementen (selektives Aendern einer Liste) und natürlich das Durchlaufen einer Liste. Wir untersuchen zuerst das Einfügen in eine Liste (list insertion).

Nehmen wir an, dass ein durch einen Zeiger (Variable) q bezeichnetes Element in eine Liste hinter dem durch den Zeiger p bezeichneten Element einzufügen sei. Die notwendigen Zeigerzuweisungen sind in (4.14) ausgeführt, und ihre Wirkung ist in Fig. 4.7 skizziert:

```
q↑.next := p↑.next; p↑.next := q                    (4.14)
```

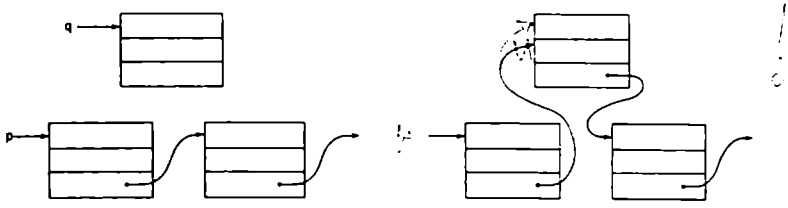


Fig. 4.7: Einfügen eines Listen-Elementes nach p↑

Soll ein Einschub vor anstatt nach dem bezeichneten Element p↑ erfolgen, dann scheint die Verbindung in einer Richtung ein Problem zu sein, da es keinerlei Weg zum Vorgänger eines Elementes gibt. Ein einfacher "Trick" löst aber unser Dilemma: Er wird in (4.15) ausgeführt und in Fig. 4.8 illustriert. Dabei wird der Schlüssel des neuen Elementes als k = 8 angenommen.

```
new(q); q↑ := p↑;                                   (4.15)
p↑.key := k; p↑.next := q
```

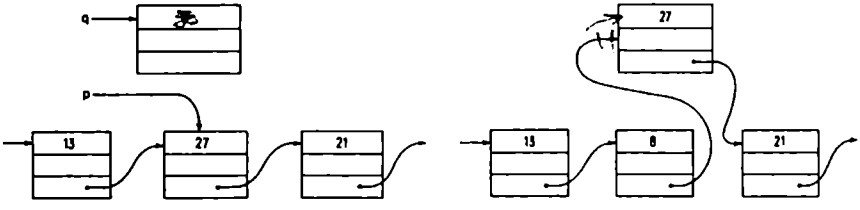


Fig. 4.8: Einfügen eines Listen-Elementes vor p↑

Der "Trick" besteht offensichtlich im Einschieben einer neuen Komponente nach p↑, wobei aber dann die Werte des neuen Elementes mit denen von p↑ vertauscht werden.

Nun betrachten wir den Prozess des Löschens in einer Liste (list deletion). Das Löschen des Nachfolgers von p↑ ist direkt ausführbar. In (4.16) ist es gezeigt in Kombination mit dem Wiedereinfügen des eliminierten Elementes am Kopf einer anderen Liste (bezeichnet durch q). r sei eine Hilfs-Variable vom Typ ↑T.

```
r := p↑.next; p↑.next := r↑.next;                    (4.16)
r↑.next := q; q := r
```

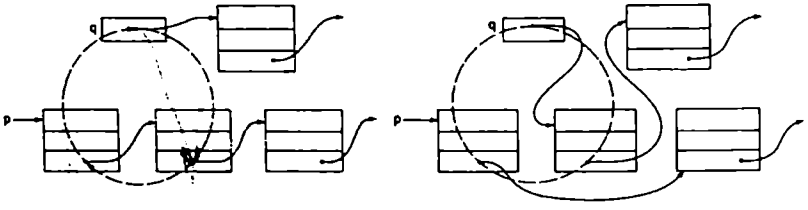


Fig. 4.9: Entfernen und Wiedereinfügen eines Listen-Elementes

Fig. 4.9 illustriert den Prozess (4.16) und macht deutlich, dass er aus einer zyklischen Vertauschung von drei Zeigern besteht.

Schwieriger ist wiederum das Entfernen eines bezeichneten Elementes selbst (anstelle seines Nachfolgers), da wir demselben Problem wie beim Einfügen vor einem p↑ begegnen: Rückverfolgung der Kette zum Vorgänger des bezeichneten Elementes ist unmöglich. Aber das Löschen des Nachfolgers, nachdem sein Wert nach vorne kopiert wurde, ist eine relativ offensichtliche und einfache Lösung. Sie kann angewendet werden, sofern p↑ einen Nachfolger hat, d.h. nicht das letzte Element der Liste ist.

Wir wenden uns nun der grundlegenden Operation des Durchlaufens einer Liste zu. Nehmen wir an, dass eine Operation P(x) auszuführen ist für jedes Element der Liste, deren erstes Element p↑ ist. Diese

Aufgabe lässt sich folgendermassen ausdrücken:

```
while "durch p bezeichnete Liste nicht leer" do
begin "führe Operation P aus";
   "gehe weiter zum Nachfolger"
end
```

Im einzelnen wird diese Operation durch Anweisung (4.17) beschrieben.

```
while p <> nil do                                        (4.17)
begin P(p↑); p := p↑.next
end
```

Aus den Definitionen der while-Anweisung und der verketteten Struktur folgt, dass P auf alle Elemente der Liste und auf keine anderen angewendet wird.

Eine sehr häufig auf Listen ausgeführte Operation ist das Suchen eines Elementes mit einem gegebenen Schlüssel. Wie bei File-Strukturen ist diese Suche rein sequentiell. Die Suche ist beendet, wenn entweder ein Element gefunden wurde oder wenn das Ende der Liste erreicht ist. Wir nehmen wiederum an, dass der Kopf der Liste durch den Zeiger p bezeichnet ist. Ein erster Versuch zur Formulierung dieser einfachen Suche ist (4.18).

```
while (p <> nil) and (p↑.key <> x) do p := p↑.next       (4.18)
```

Es ist zu beachten, dass aus p = nil folgt, dass p↑ nicht existiert. Damit kann die Auswertung der Terminationsbedingung zu einem Zugriff zu einer nicht existierenden Variablen führen (im Gegensatz zu einer Variablen mit undefiniertem Wert) und einen Fehler bei der Programmausführung verursachen. Dieser kann entweder behoben werden durch Verwendung einer expliziten Unterbrechung der Schleife, ausgedrückt durch eine goto-Anweisung (4.19), oder durch Einführen einer Hilfs-Variablen, die angibt, ob der gewünschte Schlüssel gefunden wurde (4.20).

```
while p <> nil do                                        (4.19)
   if p↑.key = x then goto Found
                 else p := p↑.next
```

Die Verwendung der goto-Anweisung bedingt die Existenz einer Marke an irgendeiner Stelle; man beachte, dass die Unverträglichkeit mit der while-Anweisung unterstrichen wird durch die Tatsache, dass die while-Klausel irreführend wird: die kontrollierte Anweisung wird nicht notwendigerweise so lange ausgeführt wie p <> nil.

```
b := true;                                                    (4.20)
while (p<>nil) and b do
   if p↑.key = x then b := false
                 else p := p↑.next
[(p=nil) or not b]
```

4.3.2. Geordnete Listen und Neuordnung von Listen

Der Algorithmus (4.20) erinnert stark an die Routinen für das Durchsuchen eines Array oder eines Files. In der Tat ist ein File nichts anderes als eine lineare Liste, in der die Technik der Verbindung zum Nachfolger nicht spezifiziert oder impliziert ist. Da die primitiven File-Operatoren weder das Einfügen neuer Elemente (ausgenommen am Ende) noch das Löschen (ausgenommen aller Elemente) erlauben, bleibt dem Entwickler eines Compilers für die Wahl der Darstellung ein weites Feld, und er kann sehr wohl sequentielle Anordnungen verwenden, bei der aufeinanderfolgende Komponenten in zusammenhängenden Speicherbereichen bleiben. Lineare Listen mit expliziten Zeigern erlauben grössere Flexibilität und sollten daher immer dann verwendet werden, wenn diese zusätzliche Beweglichkeit benötigt wird.

Als Beispiel wählen wir ein Problem, das im Laufe dieses Kapitels immer wieder zur Illustration verschiedener Lösungen und Techniken verwendet wird. Es ist das Problem des Lesens eines Textes, des Sammelns aller seiner Worte und des Zählens der Häufigkeit ihres Auftretens, d.h. das Aufstellen eines Häufigkeitsindex.

Eine offensichtliche Lösung ist die Erstellung einer Liste der im Text gefundenen Worte. Für jedes Wort wird die Liste durchsucht. Wird das Wort gefunden, so wird der Zähler der Häufigkeit erhöht, sonst wird das Wort in die Liste eingefügt. Wir nennen diesen Prozess der Einfachheit halber Suche (search), obwohl er ebenso ein

Einfügen enthalten kann. Um unsere Aufmerksamkeit auf den wesentlichen Teil der Behandlung von Listen konzentrieren zu können, nehmen wir an, dass die Worte bereits aus dem zu untersuchenden Text herausgegriffen wurden, als ganze Zahlen kodiert sind und in Form eines Eingabe-Files f zur Verfügung stehen. Die Formulierung der Prozedur search erfolgt direkt aus (4.20). Die Variable root bezieht sich auf den Kopf der Liste, wo neue Worte entsprechend (4.20) eingefügt werden. Der ganze Algorithmus ist als Programm 4.1 gelistet, das auch eine Routine zur tabellarischen Darstellung des konstruierten Häufigkeitsindex enthält. Der Prozess der Darstellung ist Beispiel für eine Aktion, die einmal für jedes Element der Liste ausgeführt wird, wie es in (4.17) schematisch dargestellt ist.

```
program list(input,output);
{direktes Einfügen in eine Liste}
   type ref = ↑word;
      word = record key: integer;
                    count: integer;
                    next: ref
             end ;
   var k: integer; root: ref;

   procedure search(x: integer; var root: ref);
      var w: ref; b: boolean;
   begin w := root; b := true;
      while (w <> nil) and b do
         if w↑.key = x then b := false else w := w↑.next;
      if b then
      begin {neuer Eintrag} w := root; new(root);
         with root↑ do
         begin key := x; count := 1; next := w
         end
      end else
      w↑.count := w↑.count + 1
   end {search} ;

   procedure printlist(w: ref);
   begin while w <> nil do
         begin writeln(w↑.key, w↑.count);
```

```
                w := w↑.next
            end
        end {printlist} ;

    begin root := nil; read(k);
        while k <> 0 do
            begin search(k, root); read(k)
            end ;
        printlist(root)
    end .
```

Programm 4.1

Der lineare Durchsuchungs-Algorithmus von Programm 4.1 erinnert an die Such-Prozedur für Arrays und Files und speziell an die Technik zur Vereinfachung der Abbruchbedingung für Schleifen: die Verwendung einer Marke. Eine Marke kann auch beim Suchen in Listen verwendet werden; sie wird durch ein leeres Element am Ende der Liste dargestellt. Die neue Prozedur ist (4.21), die die Such-Prozedur von Programm 4.1 ersetzt, wenn zusätzlich eine globale Variable sentinel vereinbart wird und die Initialisierung von root ersetzt wird durch die Anweisungen

```
new(sentinel); root := sentinel
```

die das als Marke dienende Element generieren.

```
procedure search(x: integer; var root: ref);                    (4.21)
    var w: ref;
begin w := root; sentinel↑.key := x;
    while w↑.key <> x do w := w↑.next;
    if w <> sentinel then w↑.count := w↑.count + 1 else
    begin {neuer Eintrag} w := root; new(root);
        with root↑ do
        begin key := x; count := 1; next := w
        end
    end
end {search}
```

Offensichtlich werden die Möglichkeiten und die Flexibilität der verketteten Listen in diesem Beispiel missbraucht, und das lineare Durchlaufen der ganzen Liste kann nur in den Fällen zugelassen

werden, in denen die Zahl der Elemente beschränkt ist. Eine leichte Verbesserung aber liegt auf der Hand: das Durchsuchen einer geordneten Liste. Wenn die Liste geordnet ist (etwa nach wachsenden Schlüsseln), dann kann die Suche spätestens beim Entdecken des ersten Schlüssels beendet werden, der grösser ist als der neue. Das Ordnen der Liste wird erreicht durch Einfügen neuer Elemente an der entsprechenden Stelle anstatt am Kopf. Tatsächlich ergibt sich die Ordnung praktisch ohne Aufwand. Dies kommt durch die Einfachheit des Einfügens in eine verkettete Liste, d.h. durch volle Ausnutzung ihrer Flexibilität. Diese Möglichkeit gibt es bei den Array- und File-Strukturen nicht. (Man beachte aber, dass auch in geordneten Listen kein Aequivalent zum binären Durchsuchen von Arrays existiert.)

Durchsuchen von geordneten Listen ist ein typisches Beispiel der in (4.15) diskutierten Situation, in der ein Element vor einem gegebenen eingefügt werden muss, nämlich vor dem ersten mit einem grösseren Schlüssel. Die hier verwendete Technik unterscheidet sich aber von der in (4.15); anstatt die Werte zu kopieren, werden zwei Zeiger beim Durchlaufen der Liste mitgeführt; w2 hinkt einen Schritt hinter w1 her und zeigt deshalb auf den eigentlichen Ort, an dem der Einschub stattfinden soll, wenn w1 auf einen grösseren Schlüssel zeigt. Der allgemeine Schritt des Einfügens ist in Fig. 4.10 dargestellt.

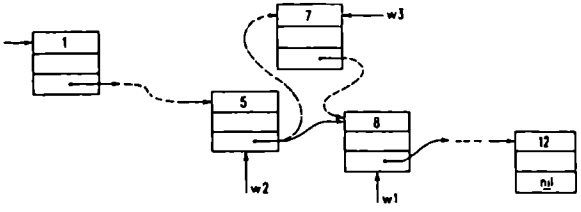


Fig. 4.10: Einfügen in eine geordnete Liste

Bevor wir weiterfahren, sind zwei Umstände zu beachten:

1. Der Zeiger des neuen Elementes (w3) muss w2↑.next zugewiesen werden, ausser wenn die Liste noch leer ist. Aus Gründen der Einfachheit und Effizienz wollen wir diese Unterscheidung nicht

durch Verwendung einer bedingten Anweisung machen. Der einzige Weg, um dies zu vermeiden, ist die Einführung eines leeren Elementes am Kopf der Liste.

2. Das Durchlaufen der Liste mit zwei sich folgenden Zeigern bedingt, dass die Liste (zusätzlich zu dem leeren) mindestens ein Element enthält. Dies bedingt, dass das Einfügen des ersten Elementes getrennt zu behandeln ist.

Ein Vorschlag, der diesen Linien und Hinweisen folgt, ist in (4.23) dargestellt. Er verwendet eine Hilfsprozedur insert (4.22), die lokal zu search zu vereinbaren ist; sie generiert und initialisiert das neue Element w.

```
procedure insert(w: ref);                                    (4.22)
   var w3: ref;
begin new(w3);
   with w3↑ do
      begin key := x; count := 1; next := w
      end ;
   w2↑.next := w3
end {insert}
```

Die Initialisierung "root := nil" in Programm 4.1 wird entsprechend ersetzt durch

```
new(root); root↑.next := nil
```

Unter Bezugnahme auf Fig. 4.10 bestimmen wir die Bedingung, unter der das Durchsuchen zum nächsten Element weitergeht; sie besteht aus zwei Faktoren, nämlich

```
(w1↑.key < x ) and (w1↑.next <> nil)
```

Damit ergibt sich die in (4.23) gezeigte Prozedur search.

```
procedure search(x: integer; var root: ref);                 (4.23)
   var w1,w2: ref;
begin w2 := root; w1 := w2↑.next;
   if w1 = nil then insert (nil) else
```

```
    begin
      while (w1↑.key < x) and (w1↑.next <> nil) do
        begin w2 := w1; w1 := w2↑.next
        end ;
      if w1↑.key = x then w1↑.count := w1↑.count + 1 else
        insert(w1)
    end
  end {search}
```

Leider enthält dieser Vorschlag trotz unserer Sorgfalt einen logischen Fehler! Dem Leser wird dringend empfohlen, das Versehen vor dem Weiterlesen zu suchen. Wer diese Aufforderung übergehen möchte, dem sei lediglich gesagt, dass (4.23) immer das zuerst eingegebene Element an das Ende der Liste stellt. Der Fehler wird korrigiert unter Berücksichtigung, dass bei Beendigung des Durchlaufens, veranlasst durch die zweite Bedingung, das neue Element nach w1↑ und nicht davor einzufügen ist. Daher ist die Anweisung "insert(w1)" zu ersetzen durch

```
  begin if w1↑.next = nil then                              (4.24)
          begin w2 := w1; w1 := nil
          end;
        insert(w1)
  end
```

Der vertrauensvolle Leser wurde hier einmal mehr gefoppt, denn (4.24) ist immer noch nicht korrekt. Um diesen Fehler einzusehen, nehme man an, dass der neue Schlüssel zwischen dem letzten und zweitletzten Schlüssel liege. Dabei werden beide Faktoren der Fortsetzungsbedingung falsch, wenn das Durchsuchen das Ende der Liste erreicht und folglich der Einschub nach dem letzten Element erfolgt. Kommt der zweite Schlüssel später erneut vor, wird er korrekt eingefügt und erscheint daher zweimal in der Tabelle. Abhilfe wird geschaffen durch Ersetzen der Bedingung

```
        w1↑.next = nil
```

in (4.24) durch

```
        w1↑.key < x .
```

Um das Durchsuchen zu beschleunigen, kann die Fortsetzungsbedingung der while-Anweisung durch Verwendung einer Marke weiter vereinfacht

werden. Dies bedingt das anfängliche Vorhandensein eines leeren Kopfes und einer Marke am Ende. Somit muss die Liste mit folgenden Anweisungen initialisiert werden:

```
new(root); new(sentinel); root↑.next := sentinel
```

und die Prozedur search in (4.25) wird merklich einfacher.

```
procedure search(x: integer; var root: ref);                    (4.25)
   var w1,w2,w3:ref;
begin w2 := root; w1 := w2↑.next; sentinel↑.key := x;
   while w1↑.key < x do
      begin w2 := w1; w1 := w2↑.next
      end ;
   if (w1↑.key = x) and (w1 <> sentinel) then
      w1↑.count := w1↑.count + 1 else
   begin new(w3); {füge w3 zwischen w1 und w2 ein}
      with w3↑ do
         begin key := x; count := 1; next := w1
         end ;
      w2↑.next := w3
   end
end {search}
```

Es ist nun an der Zeit zu fragen, welchen Gewinn wir aus den Durchsuchen geordneter Listen erwarten können. Erinnert man sich, dass praktisch keine zusätzliche Komplexität ins Spiel kam, so sollte man auch keine überwältigende Verbesserung erwarten.

Nehmen wir an, dass alle Worte im Text mit gleicher Häufigkeit vorkommen. In diesem Fall ist der Gewinn durch lexikographische Ordnung tatsächlich null, sobald alle Worte in der Liste aufgeführt sind. Denn die Position eines Wortes spielt keine Rolle, wenn nur die Summe aller Zugriffsschritte von Bedeutung ist und wenn alle Worte gleich häufig vorkommen. Ein Gewinn wird jedoch immer beim Einfügen eines neuen Wortes erzielt. Anstatt zuerst die ganze Liste zu durchsuchen, muss im Mittel nur die halbe Liste durchsucht werden. Somit lohnt sich eine geordnete Liste nur, wenn ein Häufigkeitsindex generiert werden soll, bei dem die Anzahl der verschiedenen Worte gegenüber der Häufigkeit ihres Auftretens gross

ist. Die obenstehenden Beispiele eignen sich daher eher für Programmierübungen als für praktische Anwendungen.

Die Anordnung von Daten in einer verketteten Liste wird empfohlen, wenn die Zahl der Elemente relativ klein ist (etwa kleiner als 100), sich ändert und wenn darüberhinaus nichts über die Häufigkeit des Zugriffs bekannt ist. Ein typisches Beispiel ist die Tabelle der Symbole in Compilern für Programmiersprachen. Jede Vereinbarung veranlasst den Eintrag eines neuen Symbols, und beim Verlassen dessen Gültigkeitsbereichs wird es aus der Liste gestrichen. Die Verwendung einfach verketteter Listen ist geeignet für Anwendungen mit realtiv kurzen Programmen. Aber selbst in diesem Fall kann eine beachtliche Verbesserung der Zugriffsmethode erreicht werden durch eine sehr einfache Technik, die hier erwähnt wird, hauptsächlich weil sie ein schönes Beispiel für die Flexibilität verketteter Listenstrukturen darstellt.

Programme haben die charakteristische Eigenschaft, dass das gleiche Symbol sehr oft gehäuft auftritt: d.h. einem Auftreten folgen eine oder mehrere Wiederverwendungen des gleichen Symbols. Diese Information ist eine Aufforderung zur Reorganisierung der Liste nach jedem Zugriff, indem man ein gefundenes Element an den Kopf der Liste bringt und damit die Länge des Suchweges für die nächste Suche verkürzt. Diese Zugriffsmethode heisst Durchsuchen der Liste mit Neuordnung - oder etwas hochtrabend - Durchsuchen einer selbstorganisierenden Liste. Beim Darstellen des entsprechenden Algorithmus in Form einer Prozedur, die in Programm 4.1 eingesetzt werden kann, ziehen wir Nutzen aus der bisher gemachten Erfahrung und führen gleich zu Beginn eine Marke ein. Tatsächlich beschleunigt eine Marke nicht nur die Suche, sie vereinfacht in diesem Fall sogar das Programm. Die Liste ist also zu Beginn nicht leer, sondern sie enthält bereits das Element mit der Marke. Die ersten Anweisungen sind

```
new(sentinel); root := sentinel
```

Man beachte den prinzipiellen Unterschied zwischen dem neuen Algorithmus und dem direkten Durchsuchen der Liste (4.21), der in der mit dem Finden eines Elementes verbundenen Neuordnung besteht. Das Element wird dann aus seiner alten Position entfernt und am

Anfang erneut eingefügt. Das Streichen erfordert wieder die Verwendung der beiden sich nachlaufenden Zeiger, wobei der Vorgänger w2↑ eines identifizierten Elementes w1↑ immer noch zugreifbar ist. Dies wiederum erfordert eine spezielle Behandlung des ersten Elementes (d.h. der leeren Liste). Zum Verständnis des Prozesses der Umkettung verweisen wir auf Fig. 4.11. Sie zeigt die beiden Zeiger, wenn w1↑ als das gewünschte Element erkannt ist. Die Situation nach der richtigen Neuordnung ist dargestellt in Fig. 4.12, und die neue Prozedur search ist in (4.26) zusammengefasst.

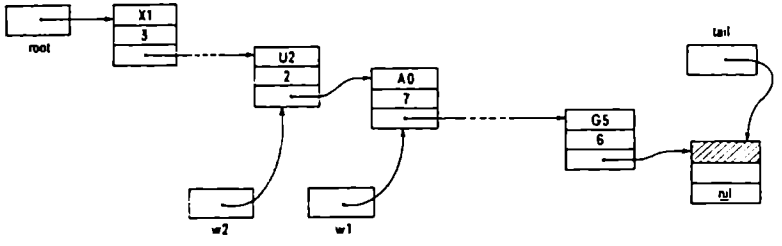


Fig. 4.11: Geordnete Liste vor dem Einfügen

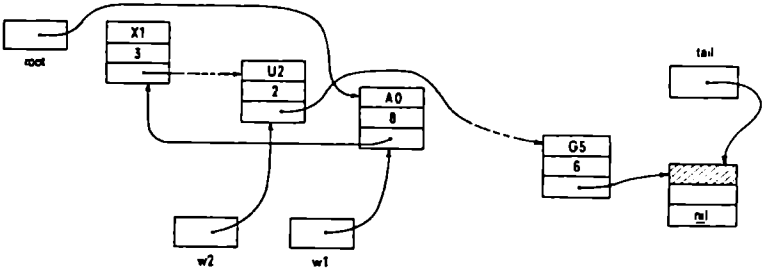


Fig. 4.12: Geordnete Liste nach dem Einfügen

```
procedure search(x: integer; var root: ref);                    (4.26)
   var w1,w2: ref;
begin w1 := root; sentinel↑.key := x;
   if w1 = sentinel then
   begin {erstes Element} new(root);
      with root↑ do
         begin key := x; count := 1; next := sentinel
```

```
            end
        end else
        if w1↑.key = x then w1↑.count := w1↑.count + 1 else
        begin {Suche}
            repeat w2 := w1; w1 := w2↑.next
            until w1↑.key = x;
            if w1 = sentinel then
            begin {Einfügen}
                w2 := root; new(root);
                with root↑ do
                    begin key := x; count := 1; next := w2
                    end
            end else
            begin {gefunden, ordne jetzt um}
                w1↑.count := w1↑.count + 1;
                w2↑.next := w1↑.next; w1↑.next := root; root := w1
            end
        end
    end {search}
```

Die Verbesserung dieser Suchmethode hängt stark vom Häufungsgrad der Eingabe-Daten ab. Bei gegebenem Häufungsfaktor kommt die Verbesserung bei grossen Listen besser zur Geltung. Um eine Vorstellung zu geben, in welcher Grössenordnung die zu erwartende Verbesserung liegt, wurde eine empirische Messung vorgenommen mit dem obigen Programm zur Bestimmung der Uebereinstimmung auf einem relativ langen Text. Dabei wurden die Methoden der Ordnung der linearen Liste (4.21) und der Neuordnung der Liste (4.26) verglichen. Die gemessenen Werte sind in Tabelle 4.2 zusammengefasst.

Programm	A	B
Anzahl versch. Schlüssel	53	582
Häufigkeit der Schlüssel	315	14'341
Zeit für Suche mit Ordnen	6'207	3'200'622
Zeit für Suche mit Neuordnen	4'529	681'584
Verbesserungsfaktor	1.37	4.70

Tabelle 4.2

Leider ist die Verbesserung am grössten, wenn ohnehin eine andere Datenorganisation notwendig ist. Wir werden auf dieses Beispiel in Abschnitt 4.4 zurückkommen.

4.3.3. Eine Anwendung: Topologisches Sortieren

Ein geeignetes Beispiel für die Verwendung einer flexiblen dynamischen Datenstruktur ist der Prozess des topologischen Sortierens. Dies ist ein Sortierprozess von Elementen, für die eine teilweise Ordnung definiert ist, d.h. es gibt eine Ordnung für einige Paare dieser Elemente, aber nicht für alle. Diese Situation kommt häufig vor; Beispiele teilweiser Ordnung sind folgende:

1. In einem Wörterbuch oder Stichwortverzeichnis werden Worte mit Hilfe anderer Worte definiert. Wenn ein Wort w definiert ist mit Hilfe eines Wortes v, so bezeichnen wir dies mit $v < w$. Topologisches Sortieren von Worten in einem Wörterbuch bedeutet die Anordnung auf solche Art, dass kein Bezug auf undefinierte Worte genommen wird.

2. Eine Aufgabe (z.B. ein technisches Projekt) wird aufgeteilt in Teilaufgaben. Gewisse Teilaufgaben müssen gewöhnlich abgeschlossen sein, bevor andere Teilaufgaben in Angriff genommen werden können. Ist eine Teilaufgabe v vor einer Teilaufgabe w auszuführen, so schreiben wir $v < w$. Topologisches Sortieren bedeutet eine derartige Anordnung, dass bei Ausführung jeder Teilaufgabe alle dazu bedingten Teilaufgaben erfüllt sind.

3. In einem Studienplan müssen gewisse Kurse vor anderen besucht werden, da sich manche Kurse im Stoff auf früher gelehrte Voraussetzungen stützen. Ist ein Kurs v Voraussetzung für einen Kurs w, so schreiben wir $v < w$. Topologisches Sortieren bedeutet eine solche Anordnung der Kurse, dass kein Kurs einen später aufgeführten Kurs voraussetzt.

4. In einem Programm können einige Prozeduren Aufrufe anderer Prozeduren enthalten. Wird eine Prozedur v durch eine Prozedur w aufgerufen, so schreiben wir $v < w$. Unter topologischem Sortieren versteht man die Anordnung der Prozedur-Vereinbarungen auf

solche Art, dass keine Vorwärtsreferenzen vorkommen.

Im allgemeinen ist die teilweise Ordnung einer Menge S eine Relation zwischen den Elementen von S. Sie wird bezeichnet durch das Symbol <, in Worten ausgedrückt durch "kommt vor", und erfüllt folgende drei Eigenschaften (Axiome) für irgendwelche verschiedene Elemente x, y und z aus S:

(1) wenn x < y und y < z, dann x < z (Transitivität)
(2) wenn x < y, dann nicht y < x (Asymmetrie) (4.27)
(3) nicht x < x (Irreflexivität)

Aus verständlichen Gründen nehmen wir an, dass die Mengen S endlich sind, die durch einen Algorithmus topologisch sortiert werden sollen. Eine teilweise Ordnung kann in Form eines Diagramms oder Graphen gezeichnet werden, wobei die Knoten die Elemente von S und die gerichteten Verbindungen die Ordnungsrelation darstellen. Fig. 4.13 gibt dazu ein Beispiel:

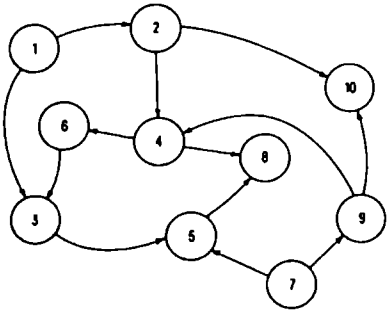


Fig. 4.13: Partiell geordnete Menge

Das Problem des topologischen Sortierens besteht darin, die teilweise Ordnung in eine lineare Ordnung einzubetten. Graphisch bedeutet dies die Anordnung der Knoten des Graphen in einer Zeile, so dass alle Pfeile wie in Fig. 4.14 nach rechts zeigen.

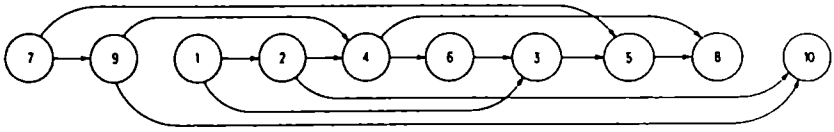


Fig. 4.14: Lineare Anordnung der Menge von Fig. 4.13

Eigenschaften (1) und (2) einer teilweisen Ordnung garantieren, dass der Graph keine Schleifen enthält. Unter genau dieser Voraussetzung ist eine solche Einbettung in eine lineare Ordnung möglich.

Wie gehen wir nun vor, um eine mögliche lineare Ordnung zu finden? Das Rezept ist sehr einfach. Wir wählen zunächst ein Element, das keinen Vorgänger hat (es muss mindestens ein solches Element geben, da sonst eine Schleife existieren würde). Dieses Objekt kommt an den Kopf der zu erstellenden Liste und wird aus der Menge S entfernt. Die restliche Menge ist immer noch teilweise geordnet, und der gleiche Algorithmus kann erneut angewendet werden, bis die Menge schliesslich leer ist.

Um diesen Algorithmus genauer zu beschreiben, müssen wir uns für eine Datenstruktur und Darstellung von S sowie der zugehörigen Ordnung entscheiden. Die Wahl der Darstellung wird durch die auszuführenden Operationen bestimmt, speziell durch die Operation des Auswählens eines Elementes ohne Vorgänger. Jedes Element sollte deshalb durch drei Eigenschaften dargestellt werden: seinen Identifikationsschlüssel, die Menge seiner Nachfolger und einen Zähler seiner Vorgänger. Da die Zahl n der Elemente in S nicht von vornherein gegeben ist, lässt sich die Menge als verkettete Liste geeignet darstellen. Folglich enthält die Beschreibung jedes Elementes zusätzlich eine Verbindung zum nächsten Element in der Liste. Wir wollen annehmen, dass die Schlüssel ganzzahlig sind (aber nicht notwendigerweise aus den ganzen Zahlen von 1 bis n bestehen). Entsprechend lässt sich die Menge der Nachfolger jedes Elementes geeignet durch eine verkettete Liste darstellen. Jedes Element in der Liste der Nachfolger ist durch eine Identifikation und eine Verbindung zum nächsten Element in dieser Liste beschrieben. Wenn wir mit <u>leader</u> die Deskriptoren der Hauptliste bezeichnen, in der jedes Element von S genau einmal vorkommt, und mit <u>trailer</u> die Elemente der Listen der Nachfolger, so erhalten wir die folgenden

Vereinbarungen für die Datentypen:

```
type lref = ↑leader;                                        (4.28)
     tref = ↑trailer;
     leader = record key, count: integer;
                     trail: tref;
                     next: lref
              end;
     trailer = record id: lref;
                      next: tref
               end
```

Wir gehen davon aus, dass die Menge und ihre Ordnungsrelation ursprünglich als Folge von Paaren der Schlüssel auf dem Eingabe-File dargestellt sind. Die Eingabe-Daten für das Beispiel von Fig. 4.13 sind in (4.29) gezeigt, wobei die Symbole < der besseren Uebersicht halber hinzugefügt wurden.

```
1 < 2,   2 < 4,   4 < 6,   2 < 10,   4 < 8,          (4.29)
6 < 3,   1 < 3,   3 < 5,   5 < 8,    7 < 5,
7 < 9,   9 < 4,   9 < 10
```

Der erste Teil des Programms zur topologischen Sortierung muss das Eingabe-File lesen und die Daten in die entsprechende Listenstruktur bringen. Dies geschieht durch fortgesetztes Lesen eines Schlüsselpaares x und y (x < y). Die Zeiger zu ihren Darstellungen in der verketteten Liste der Elemente vom Typ leader bezeichnen wir mit p und q. Diese Records müssen zuerst in der Liste gesucht werden und, falls nicht vorhanden, in die Liste eingefügt werden. Diese Aufgabe wird durch eine Funktions-Prozedur mit Namen L ausgeführt. Dann wird ein neues Element in der Liste der trailer von x eingefügt, versehen mit der Identifikation von y; der Zähler der Vorgänger von y wird um 1 erhöht. Dieser Algorithmus heisst "Eingabe-Phase" (4.30). Fig. 4.15 illustriert die Datenstruktur, die beim Bearbeiten der Eingabe-Daten (4.29) durch (4.30) generiert wird. Dieses Programmstück bezieht sich auf eine Funktion L(w), die die Referenz zur Listenkomponente mit Schlüssel w liefert (siehe auch Programm 4.2). Wir nehmen an, dass die Sequenz der Eingabe der Schlüsselpaare durch eine zusätzliche Null abgeschlossen wird.

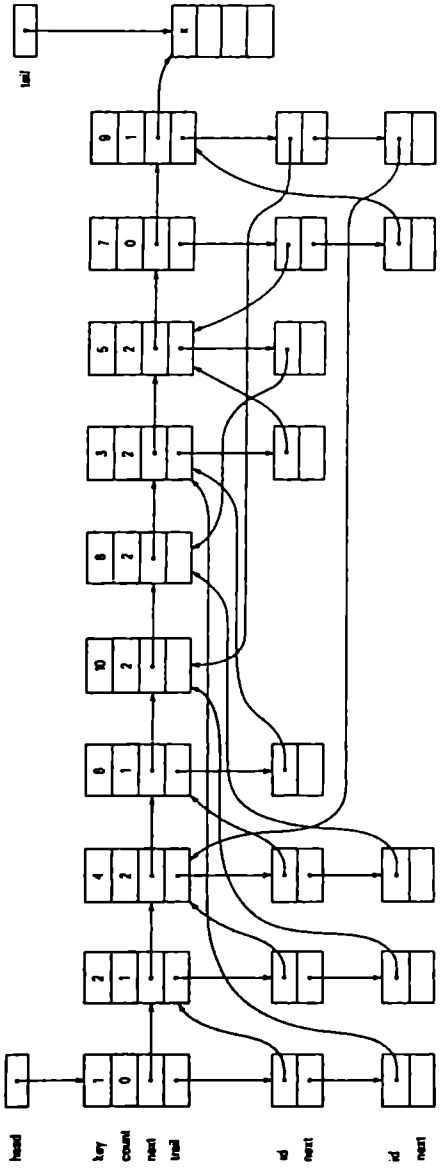


Fig. 4.15: Beim topologischen Sortieren erzeugte Listenstruktur

```
[Eingabe-Phase] read(x);                                    (4.30)
new(head); tail := head; z := 0;
while x <> 0 do
begin read(y); p := L(x); q := L(y);
   new(t); t↑.id := q; t↑.next := p↑.trail;
   p↑.trail := t; q↑.count := q↑.count + 1;
   read(x)
end
```

Nach Konstruktion der Datenstruktur von Fig. 4.15 in diesem Eingabeteil kann der eigentliche Prozess des topologischen Sortierens, so wie er oben beschrieben wurde, aufgenommen werden. Da er aber aus dem fortgesetzten Auswählen eines Elementes besteht, dessen Zähler der Vorgänger Null ist, scheint es angebracht, alle diese Elemente in einer Kette zusammenzufassen. Wie wir sehen, wird die ursprüngliche Kette der leader später nicht mehr benötigt, so dass das gleiche Feld mit Namen next erneut verwendet werden kann, um die Elemente ohne Vorgänger zu verbinden. Diese Operation des Ersatzens einer Kette durch eine andere kommt in der Listenverarbeitung häufig vor. Sie wird in (4.31) im einzelnen beschrieben und erstellt aus Bequemlichkeitsgründen eine neue Katte in umgekehrter Richtung.

```
[suche leader ohne Vorgänger]                               (4.31)
   p := head; head := nil;
   while p <> tail do
      begin q := p; p := q↑.next;
           if q↑.count = 0 then
           begin {füge q↑ in neue Kette ein}
                q↑.next := head; head := q
           end
   end
```

In bezug auf Fig. 4.15 wird die in next verwirklichte Kette der leader ersetzt durch eine in Fig. 4.16 dargestellte, wobei die nicht gezeichneten Zeiger unverändert bleiben.

Nach dieser vorbereitenden Aufstellung einer geeigneten Darstellung der teilweise geordneten Menge S können wir schliesslich zum Problem der topologischen Sortierung kommen, d.h. zur Erstellung

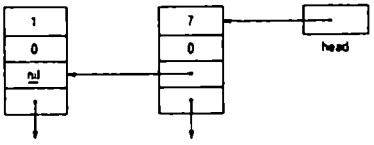


Fig. 4.16: Liste der Listenanfänge mit
Zählerwerten = 0

der Ausgabe-Sequenz. In einem ersten Entwurf kann die Lösung folgendermassen beschrieben werden:

```
q := head;                                                    (4.32)
while q <> nil do
begin {drucke dieses Element, lösche es dann}
     writeln(q↑.key); z := z-1;
     t := q↑.trail; q := q↑.next;
     "erniedrige den Zähler der Vorgänger für alle seine
      Nachfolger in der Liste t der trailer; wird ein
      Zähler 0, ist dieses Element in die Liste q der leader
      einzufügen"
end
```

Die in (4.32) zu verfeinernde Anweisung besteht einmal mehr in Durchsuchen einer Liste (siehe Schema (4.17)). Bei jedem Schritt bezeichnet die Hilfs-Variable p das Element vom Typ leader, dessen Zähler vermindert und getestet werden soll.

```
while t <> nil do                                             (4.33)
begin p := t↑.id; p↑.count := p↑.count-1;
      if p↑.count = 0 then
      begin  {füge p↑ in die Liste der leader ein}
           p↑.next := q; q:= p
      end;
      t := t↑.next
end
```

Damit ist das Programm für die topologische Sortierung vollständig. Man beachte, dass ein Zähler z eingeführt wurde, der die Elemente vom Typ leader im Eingabeteil gezählt hat. Dieser Zähler wird nun

jedesmal vermindert, wenn ein Element im Ausgabeteil ausgegeben wird. Am Ende des Programms sollte er deshalb wieder den Wert Null haben. Ist dies nicht der Fall, so sind Elemente in der Struktur zurückgeblieben, die alle einen Vorgänger haben. In diesem Fall ist die Menge S offensichtlich nicht partiell geordnet. Wir vervollständigen deshalb das Programm, das jetzt aus einem Eingabe-, Such- und Ausgabeteil besteht mit der Anweisung

```
if z <> ∅ then write(" Menge nicht teilweise geordnet")
```

Der oben programmierte Ausgabeteil ist Beispiel eines Prozesses, der eine pulsierende Liste unterhält, d.h. es werden Elemente in nicht vorhersagbarer Reihenfolge eingefügt und gelöscht. Er ist deshalb ein Beispiel für einen Prozess, der die Beweglichkeit der explizit verketteten Listen voll ausschöpft.

```
program topsort(input,output);
type lref = ↑leader;
   tref = ↑trailer;
   leader = record key: integer;
               count: integer;
               trail: tref;
               next:  lref
            end ;
   trailer= record id: lref;
               next: tref
            end ;
var head, tail, p, q: lref;
   t: tref; z: integer;
   x,y: integer;

function L(w: integer): lref;
   {Referenz zu leader mit Schlüssel w}
   var h: lref;
begin h := head; tail↑.key := w;  {Marke}
   while h↑.key <> w do  h := h↑.next;
   if h = tail then
      begin {kein Element mit Schlüssel w in der Liste}
         new(tail); z := z+1;
         h↑.count := ∅; h↑.trail := nil; h↑.next := tail
```

```
      end ;
   L := h
end {L} ;

begin {Initialisiere Liste der leader mit Pseudo-Element}
   new(head); tail := head; z := 0;
{Eingabe-Phase}  read(x);
   while x <> 0 do
   begin read(y); writeln(x,y);
      p := L(x); q := L(y);
      new(t); t↑.id := q; t↑.next := p↑.trail;
      p↑.trail := t; q↑.count := q↑.count + 1;
      read(x)
   end ;
{Suche leader mit count = 0}
   p := head; head := nil;
   while p <> tail do
   begin q := p; p := p↑.next;
      if q↑.count = 0 then
         begin q↑.next := head; head := q
         end ;
   end ;
{Ausgabe-Phase}  q := head;
   while q <> nil do
   begin writeln(q↑.key); z := z-1;
      t := q↑.trail; q := q↑.next;
      while t <> nil do
      begin p := t↑.id; p↑.count := p↑.count - 1;
         if p↑.count = 0 then
         begin {füge p↑ in die q-list ein}
            p↑.next := q; q := p
         end ;
         t := t↑.next
      end
   end ;
   if z <> 0 then
      writeln(' Diese Menge ist nicht partiell geordnet')
end .
```

Programm 4.2

4.4. Baumstrukturen

4.4.1. Grundlegende Konzepte und Definitionen

Wir haben gesehen, dass Sequenzen und Listen auf folgende Art zweckmässig definiert werden können: Eine Sequenz (Liste) mit zugrunde liegendem Typ T ist entweder

1. die leere Sequenz (Liste) oder
2. die Zusammensetzung (Verkettung) eines T mit einer Sequenz vom Grundtyp T.

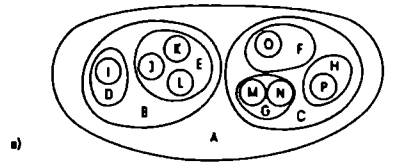
Dabei wird die Rekursion als Hilfsmittel zur Definition eines Prinzips der Strukturierung verwendet, nämlich der Sequenz oder Wiederholung. Sequenzen und Wiederholungen sind so gebräuchlich, dass sie gewöhnlich als grundlegende Muster der Struktur und des Ablaufs betrachtet werden. Man sollte aber nicht vergessen, dass sie mittels Rekursion definiert werden können, ~~während die Umkehrung nicht gilt~~. Denn Rekursion kann effizient und sauber zur Definition viel komplizierterer Strukturen verwendet werden. Bäume sind ein bekanntes Beispiel. Eine Baumstruktur sei folgendermassen definiert: Eine Baumstruktur mit Grundtyp T ist entweder

1. die leere Struktur oder
2. ein Knoten vom Typ T mit einer endlichen Zahl verknüpfter, voneinander verschiedener Baumstrukturen mit Grundtyp T, sogenannter Teilbäume (subtree).

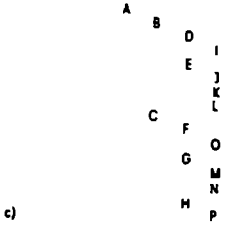
Aus der Aehnlichkeit der rekursiven Definition von Sequenzen und Baumstrukturen ergibt sich, dass die Sequenz (Liste) eine Baumstruktur ist, in der jeder Knoten höchstens einen Teilbaum hat. Die Sequenz (Liste) heisst deshalb auch entarteter Baum (degenerate tree).

Es gibt verschiedene Wege zur Darstellung einer Baumstruktur. Erstreckt sich der Grundtyp T über die Buchstaben, so zeigt Fig. 4.17 verschiedene Darstellungen einer Baumstruktur. Diese Darstellungen zeigen alle die gleiche Struktur und sind deshalb äquivalent. Es ist die Struktur des Graphen, die anschaulich die Verzweigungen

zeigt und daher zum allgemein gebräuchlichen Namen "Baum" (tree) Anlass gegeben hat.



b) (A(B(D(I)), E(J,K,L)),C(F(O),G(M,N),H(P)))



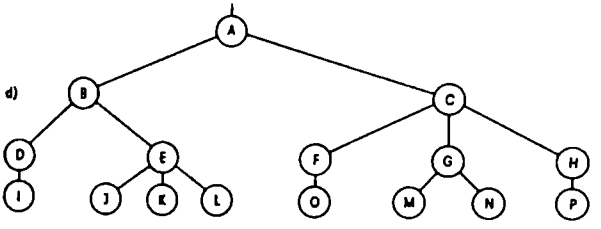


Fig. 4.17: Darstellung von Baumstrukturen durch
(a) geschachtelte Mengen,
(b) geschachtelte Klammern,
(c) Einrückung, (d) Graphen

Seltsamerweise werden die Bäume gewöhnlich von oben nach unten gezeichnet oder - anders gesagt - die Wurzeln der Bäume zeigen nach oben. Die letzte Formulierung ist aber irreführend, da der oberste Knoten (a) allgemeinhin die Wurzel (root) heisst. Obwohl wir wissen, dass Bäume in Wirklichkeit kompliziertere Geschöpfe sind

als unsere Abstraktionen, wollen wir unsere Baumstrukturen im folgenden einfach Bäume nennen.

Ein geordneter Baum ist ein Baum, dessen Verzweigungen in jedem Knoten geordnet sind. Somit sind die zwei geordneten Bäume in Fig. 4.18 verschieden. Ein Knoten y, der direkt unter einem Knoten x liegt, heisst (direkter) Nachfolger (descendant) von x; ist x auf der Stufe i, so nennt man y von der Stufe i+1. Umgekehrt heisst der Knoten x direkter Vorgänger (ancestor) von y. Die Wurzel eines Baumes liegt nach Definition auf Stufe 1. Die grösste Stufe eines Elementes des Baumes heisst seine Tiefe oder Höhe.

Fig. 4.18: Zwei verschiedene binäre Bäume

Hat ein Element keine Nachfolger, so heisst es Endelement oder Blatt (leaf); ein Element, das nicht Endelement ist, ist ein innerer Knoten. Die Zahl der (direkten) Nachfolger eines inneren Knotens heisst sein Grad. Der höchste Grad unter allen Knoten ist der Grad des Baumes. Die Zahl der Verzweigungen oder Verbindungen, die bis zum Erreichen eines Knotens x von der Wurzel aus zu durchlaufen sind, heisst die Weglänge von x. Die Wurzel hat Weglänge 1, ihre direkten Nachfolger haben Weglänge 2, usw. Allgemein hat ein Knoten auf Stufe i die Weglänge i. Die Weglänge eines Baumes ist definiert als die Summe der Weglängen aller seiner Komponenten. Sie heisst auch seine innere Weglänge. Die innere Weglänge des Baumes in Fig. 4.17 z.B. ist 52. Offensichtlich gilt für die mittlere Weglänge PI

$$PI = (1/n) * \sum_{i} n[i] * i \qquad (4.34)$$

wobei n[i] die Zahl der Knoten auf Stufe i angibt. Zur Definition der äusseren Weglänge erweitern wir den Baum überall dort um einen speziellen Knoten, wo ein leerer Teilbaum im ursprünglichen Baum vorhanden war. Dabei nehmen wir an, dass alle Knoten vom gleichen Grad sind, nämlich vom Grad des Baumes. Die Erweiterung des Baumes

auf diese Art dient daher dem Auffüllen leerer Zweige, wobei die speziellen Knoten natürlich keine weiteren Nachfolger haben. Der Baum von Fig. 4.17, erweitert mit den speziellen Knoten, ist in Fig. 4.19 dargestellt, in der die speziellen Knoten durch Quadrate dargestellt sind.

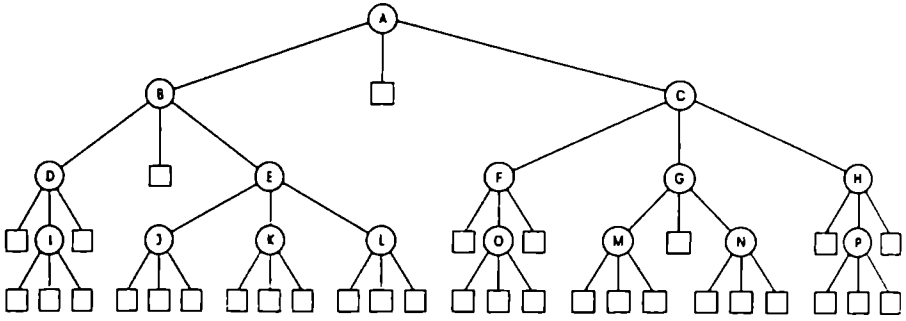


Fig. 4.19: Ternärer Baum,
durch "äussere" Elemente erweitert

Die äussere Weglänge wird nun definiert als Summe der Weglängen über alle speziellen Knoten. Wenn m[i] die Anzahl der speziellen Knoten auf Stufe i darstellt, so ergibt sich die mittlere äussere Weglänge PE zu

$$PE = (1/m) * \sum_{i} m[i] * i \qquad (4.35)$$

Für den in Fig. 4.19 dargestellten Baum ist die äussere Weglänge gleich 153.

Die Anzahl m der speziellen Knoten, die zu einem Baum vom Grad d hinzuzufügen ist, hängt direkt von der Anzahl n der ursprünglichen Knoten ab. Man beachte, dass zu jedem Knoten genau eine Verbindung führt. Somit gibt es m+n Verbindungen in dem erweiterten Baum. Auf der anderen Seite gehen d Verbindungen von jedem ursprünglichen Knoten aus, keine von den speziellen Knoten. Daher existieren d*n+1 Verbindungen, wobei die 1 von der zur Wurzel zeigenden Verbindung herrührt. Die beiden Resultate führen zur folgenden Gleichung zwischen der Zahl m der speziellen Knoten und n der ursprünglichen Knoten: d*n+1 = m+n, bzw.

$$m = (d-1)n+1 \ . \qquad (4.36)$$

Die Höchstzahl an Knoten in einem Baum mit gegebener Höhe h ist erreicht, wenn alle Knoten d Teilbäume haben, ausgenommen auf der Stufe h, wo alle Knoten keine Teilbäume haben. Für einen Baum vom Grad d, enthält Stufe 1 einen Knoten (nämlich die Wurzel), Stufe 2 enthält die d Nachfolger, Stufe 3 enthält die d**2 Nachfolger der d Knoten auf Stufe 2, usw. Dies führt zu

$$N(d,h) = 1 + d + d**2 + \ldots + d**(h-1) \qquad (4.37)$$

als Höchstzahl von Knoten für einen Baum mit Höhe h und Grad d. Für d = 2 erhalten wir

$$N(2,h) = \sum_{i=0}^{h-1} 2**i = 2**h - 1 \qquad (4.38)$$

Von besonderer Bedeutung sind geordnete Bäume vom Grad 2. Sie heissen binäre Bäume (binary trees). Wir definieren einen binären Baum als eine endliche Menge von Elementen (Knoten), die entweder leer ist oder aus einer Wurzel (Knoten) mit zwei verschiedenen binären Bäumen besteht, dem linken und rechten Teilbaum der Wurzel. In den folgenden Abschnitten werden wir ausschliesslich binäre Bäume behandeln und deshalb das Wort "Baum" in der Bedeutung "geordneter binärer Baum" verwenden. Bäume mit Grad grösser als 2 heissen Vielweg-Bäume und werden in Abschnitt 4.5 betrachtet.

Bekannte Beispiele binärer Bäume sind der Familienbaum (Stammbaum) mit Vater und Mutter einer Person als deren Nachfolger (!), die Aufzeichnung eines Tennistourniers, in der jedes Spiel durch einen Knoten mit dem Namen des Gewinners und die beiden vorausgehenden Spiele der Gegner als dessen Nachfolger aufgeführt sind, oder ein arithmetischer Ausdruck mit dyadischen Operatoren, wobei jeder Operator eine Verzweigung darstellt mit den zugehörigen Operanden als Teilbäume (s. Fig. 4.20).

Wenden wir uns nun dem Problem der Darstellung von Bäumen zu. Die Illustration solcher rekursiven Strukturen durch Verzweigungsstruk-

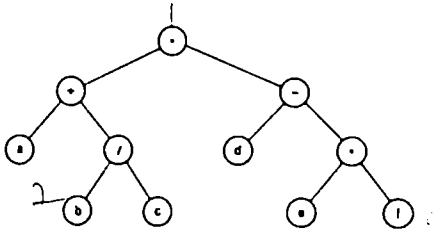


Fig. 4.20: Der arithmetische Ausdruck (a+b)/c)*(d-e*f)
dargestellt als Baum

turen legt sofort die Verwendung unserer Zeiger nahe. Es ist sicher nicht sinnvoll, Variable mit einer festen Baumstruktur zu definieren; wir definieren vielmehr die Knoten (node) als Variable mit einer festen Struktur, d.h. mit einem festen Typ, wobei der Grad des Baumes die Zahl der Zeigerkomponenten angibt, die zu Teilbäumen des Knotens zeigen. Offensichtlich wird eine Referenz auf den leeren Baum durch nil bezeichnet. Somit besteht der Baum von Fig. 4.20 aus Komponenten von folgendem Typ

```
type node = record op: char;                    (4.39)
                   left,right: ↑node
            end
```

und kann wie in Fig. 4.21 konstruiert werden.

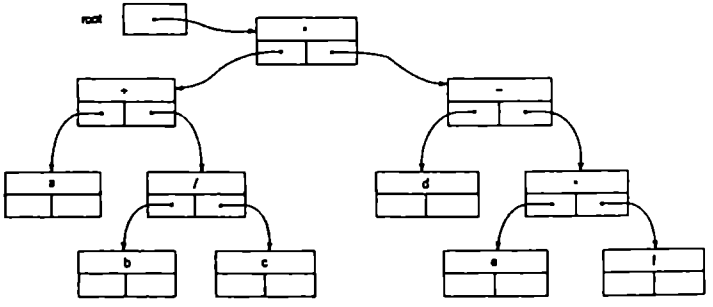


Fig. 4.21: Baum von Fig. 4.20 dargestellt
als verkettete Struktur

Es gibt sicher Wege, um die abstrakte Idee der Baumstruktur durch andere zur Verfügung stehende Datentypen, etwa Arrays, darzustellen. Dies ist in allen Sprachen üblich, die nicht die Möglichkeit der dynamischen Zuweisung von Komponenten und den Zugriff zu ihnen durch Zeiger zur Verfügung stellen. In diesem Fall könnte der Baum von Fig. 4.20 durch eine folgendermassen definierte Array-Variable dargestellt werden

```
t: array [1..11] of                                    (4.40)
        record op: char;
               left,right: integer
        end
```

und hätte die in Tabelle 4.3 gezeigten Komponentenwerte.

1		*	2	3
2		+	6	4
3		-	9	5
4		/	7	8
5		*	10	11
6		a	0	0
7		b	0	0
8		c	0	0
9		d	0	0
10		e	0	0
11		f	0	0

Tabelle 4.3: In einem Array dargestellter Baum

Obwohl die eigentliche abstrakte Struktur der durch den Array t dargestellten Daten ein Baum ist, nennen wir dies nicht einen Baum, sondern einen Array, entsprechend der expliziten Vereinbarung. Wir wollen keine weiteren Möglichkeiten zur Darstellung von Bäumen in Systemen ohne dynamische Zuweisung erörtern, da wir annehmen, dass Programmiersysteme und -sprachen mit dieser Möglichkeit allgemein verfügbar sind oder sein werden.

Bevor wir untersuchen, wie Bäume vorteilhaft verwendet werden können und wie Operationen auf Bäumen auszuführen sind, geben wir ein Beispiel, wie ein Baum durch ein Programm konstruiert wird. Es

sei ein Baum zu generieren mit Knoten von dem in (4.39) definierten Typ, und die Werte der Knoten sollen von einem Eingabe-File einzulesende Zahlen sein. Um das Problem interessanter zu machen, soll ein Baum mit n Knoten und minimaler Höhe generiert werden.

Um eine minimale Höhe für eine gegebene Zahl von Knoten zu erhalten, muss man die grösstmögliche Zahl von Knoten auf allen Stufen mit Ausnahme der untersten anordnen. Dies wird sicher erreicht mit der Strategie, die ankommenden Knoten gleichmässig nach links und rechts bei jedem Knoten zu verteilen. Dies bedeutet, dass wir die Bäume für gegebenes n wie in Fig. 4.22 (für n = 1,...,7) strukturieren.

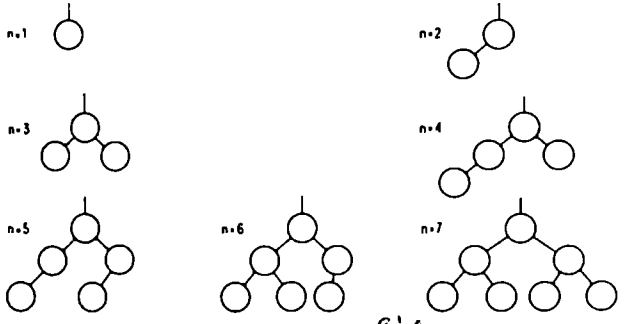


Fig. 4.22: Vollständig ausgeglichene Bäume

Die Regel für die gleichmässige Verteilung auf eine bekannte Zahl von n Knoten wird am besten rekursiv formuliert:

1. Man verwende einen Knoten für die Wurzel.
2. Man generiere den linken Teilbaum mit nl = n div 2 Knoten auf diese Art.
3. Man generiere den rechten Teilbaum mit nr = n-nl-1 Knoten auf diese Art.

Diese Regel ist ausgedrückt durch eine rekursive Prozedur als Teil von Programm 4.3, das das Eingabe-File liest und den vollständig ausgeglichenen Baum konstruiert. Wir legen folgende Definition

fest:

Ein Baum ist vollständig ausgeglichen, wenn sich für jeden Knoten die Zahlen der Knoten in seinem linken und rechten Teilbaum um höchstens 1 unterscheiden.

```
program buildtree(input,output);
type ref = ↑node;
   node = record key: integer;
             left, right: ref
          end ;
var n: integer; root: ref;

function tree(n: integer): ref;
   var newnode: ref;
      x, nl, nr: integer;
begin {Erstelle vollständig ausgeglichenen Baum mit n Knoten}
   if n = 0 then tree := nil else
   begin nl := n div 2; nr := n-nl-1;
      read(x); new(newnode);
      with newnode↑ do
         begin key := x; left := tree(nl); right := tree(nr)
         end ;
      tree := newnode
   end
end {tree} ;

procedure printtree(t: ref; h: integer);
   var i: integer;
begin {drucke Baum t um h eingerückt}
   if t <> nil then
   with t↑ do
   begin printtree(left, h+1);
      for i := 1 to h do write('   ');
      writeln(key);
      printtree(right, h+1)
   end
end {printtree} ;
```

```
begin {die erste ganze Zahl ist die Zahl der Knoten}
   read(n);
   root := tree(n);
   printtree(root,0)
end .
```

Programm 4.3

Verwendet man z.B. die folgenden Eingabe-Daten

```
21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18
```

für einen Baum mit 21 Knoten, so konstruiert Programm 4.3 den in Fig. 4.23 dargestellten vollständig ausgeglichenen Baum.

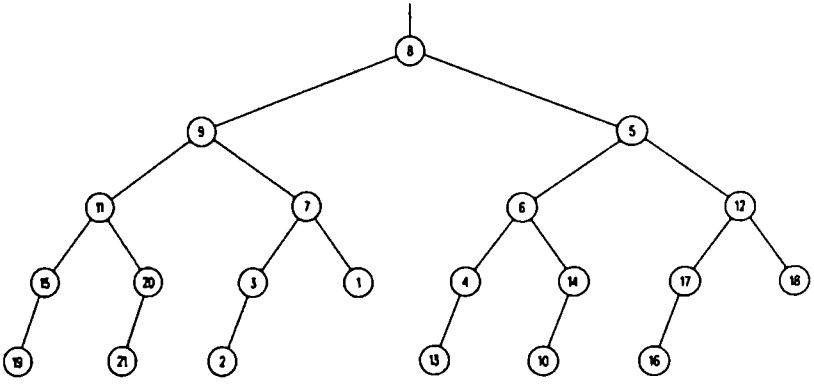


Fig. 4.23: Durch Programm 4.3 erzeugter Baum

Man beachte, wie einfach und übersichtlich dieses Programm durch Verwendung rekursiver Prozeduren wird. Offenbar sind rekursive Algorithmen besonders geeignet, wenn ein Programm Information verarbeiten soll, deren Struktur selbst rekursiv definiert ist. Dies kommt auch in der Prozedur zur Ausgabe des Resultatbaumes zum Ausdruck: Für den leeren Baum wird nichts ausgegeben, für den Teilbaum auf Stufe L wird zuerst sein eigener linker Teilbaum gedruckt, dann der Knoten, sauber eingerückt durch Voranstellen von L Leerzeichen, und schliesslich wird sein rechter Teilbaum gedruckt.

Der Vorteil des rekursiven Algorithmus wird besonders klar beim Vergleich mit einer nicht rekursiven Formulierung. Dem Leser wird dringend empfohlen, mit seinem Scharfsinn ein nicht rekursives, zu obigem äquivalentes Programm für die Erstellung eines Baumes zu schreiben, bevor er (4.41) anschaut. Dieses Programm ist ohne weiteren Kommentar angeführt.

```
program buildtree(input,output);                                    (4.41)
type ref = ↑node;
   node = record key: integer;
             left, right: ref
          end ;
var i,n,nl,nr,x: integer;
   root,p,q,r,dmy: ref;
   s: array [1..30] of    {stack}
        record n: integer; rf: ref
        end ;

procedure printtree(t: ref; h: integer);
 ...
end {printtree} ;

begin {die erste ganze Zahl ist die Zahl der Knoten}
   read(n); new(root); new(dmy);  {dummy}
   i := 1; s[1].n := n; s[1].rf := root;
   repeat n := s[i].n; r := s[i].rf; i := i-1; {pop}
      if n = 0 then r↑.right := nil else
      begin p := dmy;
         repeat nl := n div 2; nr := n-nl-1;
            read(x); new(q); q↑.key := x;
            i := i+1; s[i].n := nr; s[i].rf := q; {push}
            n := nl; p↑.left := q; p := q
         until n = 0;
         q↑.left := nil; r↑.right := dmy↑.left
      end
   until i = 0;
   printtree(root↑.right,0)
end .
```

4.4.2. Elementare Operationen auf binären Bäumen

Es gibt viele Aufgaben, die man auf einer Baumstruktur ausführen will; dabei ist häufig eine gegebene Operation P auf jedem Element des Baumes auszuführen. P ist dann aufzufassen als Parameter der allgemeineren Aufgabe des Besuchens aller Knoten oder, wie es auch oft genannt wird, des Durchlaufens eines Baumes (tree traversal).

Wenn wir die Aufgabe als einen einzigen sequentiellen Prozess betrachten, so werden die einzelnen Knoten in einer gewissen Ordnung besucht und können als linear angeordnet angenommen werden. Tatsächlich wird die Beschreibung vieler Algorithmen wesentlich einfacher, wenn wir über die Bearbeitung des "nächsten" Elementes in einem Baum sprechen können mit Bezug auf die zugrunde liegende Ordnung.

Es gibt im wesentlichen drei Ordnungen, die sich aus der Baumstruktur auf natürliche Art ergeben. Wie die Baumstruktur selbst werden sie zweckmässigerweise rekursiv dargestellt. Mit bezug auf den binären Baum in Fig. 4.24, in der W die Wurzel und A und B den linken und rechten Teilbaum bezeichnen, ergeben sich die drei Ordnungen

1. Preorder : W ,A ,B (besuche Wurzel vor den Teilbäumen)
2. Inorder : A ,W ,B
3. Postorder : A ,B ,W (besuche Wurzel nach den Teilbäumen)

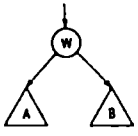


Fig. 4.24: Allgemeiner binärer Baum

Durchlaufen des Baumes von Fig. 4.20 und Aufzeichnen der Zeichen in der Reihenfolge, wie sie in den Knoten gefunden wurden, ergeben folgende Ordnungen:

1. Preorder : * + a / b c - d * e f
2. Inorder : a + b / c * d - e * f

3. Postorder : a b c / + d e f * - *

Wir erkennen die drei Formen der Ausdrücke: Preorder-Durchlauf des Ausdrucksbaumes ergibt Prefix-Notation, Postorder-Durchlauf erzeugt Postfix-Notation, und Inorder-Durchlauf erzeugt die übliche Infix-Notation, wenn auch ohne die zur Angabe der Operator-Präzedenzen notwendigen Klammern.

Wir wollen nun die drei Arten des Durchlaufens durch drei konkrete Prozeduren mit dem expliziten Parameter t zur Bezeichnung des zu bearbeitenden Baumes formulieren. Der implizite Parameter p bezeichnet die auf jedem Knoten auszuführende Operation. Wir gehen von folgenden Definitionen aus

```
type ref = ↑node;                                    (4.42)
     node = record ...
                  left,right: ref
            end
```

Die drei Methoden werden als rekursive Prozeduren formuliert, um erneut zu zeigen, dass Operationen auf rekursiv definierten Datenstrukturen am zweckmässigsten als rekursive Operatoren definiert werden.

```
procedure preorder(t: ref);                          (4.43)
begin if t <> nil then
        begin p(t);
              preorder(t↑.left);
              preorder(t↑.right)
        end
end

procedure inorder(t: ref);                           (4.44)
begin if t <> nil then
      begin inorder(t↑.left);
            p(t);
            inorder(t↑.right)
      end
end
```

```
procedure postorder(t: ref);                                    (4.45)
begin if t <> nil then
      begin postorder(t↑.left);
            postorder(t↑.right);
            p(t)
      end
end
```

Man beachte, dass der Zeiger t als Value-Parameter übergeben wird. Dies bringt zum Ausdruck, dass die wesentliche Einheit die Referenz zum betrachteten Teilbaum ist und nicht die Variable, deren Wert der Zeiger ist, der geändert werden könnte, wenn t als Variablen-Parameter übergeben würde.

Ein Beispiel für eine Routine, die einen Baum durchläuft, ist Programm 4.3, das einen Baum mit geeignetem Einrücken zur Angabe der Stufe jedes Knotens druckt.

Binäre Bäume werden oft verwendet, um eine Menge von Daten darzustellen, deren Elemente nach einem Schlüssel wiederzufinden sind. Wenn ein Baum derart organisiert ist, dass für jeden Knoten t[i] alle Schlüssel im linken Teilbaum von t[i] kleiner sind als der Schlüssel von t[i] und die im rechten Teilbaum grösser sind als der Schlüssel von t[i], dann heisst dieser Baum Suchbaum (search tree). In einem Suchbaum ist es möglich, einen beliebigen Schlüssel zu finden, indem man, ausgehend von der Wurzel, dem Suchpfad entlang zum rechten oder linken Teilbaum eines Knotens geht, wobei die Wahl der Richtung nur von dem Schlüssel dieses einen Knotens abhängt. Wie wir gesehen haben, können n Elemente in einem binären Baum mit der kleinen Höhe log(n) angeordnet werden. Somit kann eine Suche unter n Elementen mit nur log(n) Vergleichen ausgeführt werden, wenn der Baum vollständig ausgeglichen ist. Der Baum ist offensichtlich eine geeignetere Form für die Organisation einer solchen Datenmenge als die im vorangehenden Abschnitt verwendete lineare Liste.

Da diese Suche einem einzigen Weg von der Wurzel zum gewünschten Knoten folgt, kann sie leicht durch Iteration programmiert werden.

```
function loc(x: keytype; t: ref): ref;                    (4.46)
   var found: boolean;
begin found := false;
   while (t <> nil) and not found do
   begin
      if t↑.key = x then found := true else
      if t↑.key > x then t := t↑.left else t := t↑.right
   end;
   loc := t
end
```

Die Funktion loc(x,t) liefert den Wert nil, wenn kein Schlüssel mit dem Wert x im Baum mit der Wurzel t gefunden wurde. Wie beim Suchen durch eine Liste veranlasst die Komplexität der Abbruchbedingung die Suche nach einer besseren Lösung. Sie besteht in der Verwendung einer Marke am Ende der Liste. Diese Technik ist auch im Fall eines Baumes anwendbar. Die Verwendung von Zeigern ermöglicht es, dass alle Zweige des Baumes mit der gleichen identischen Marke abschliessen. Die sich daraus ergebende Struktur ist kein Baum mehr, sondern eher ein Baum, bei dem alle Blätter mit Schnüren zu einem einzigen Verankerungspunkt verknüpft sind (Fig. 4.25). Man kann diese Marke als gemeinsamen Vertreter aller äusseren Knoten betrachten, um die der ursprüngliche Baum erweitert wurde (vgl. Fig. 4.19).

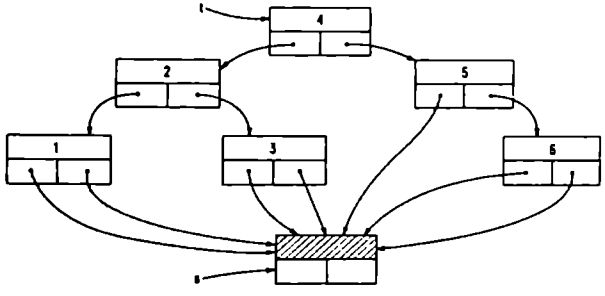


Fig. 4.25: Geordneter Baum mit "Marke"

Damit erhalten wir in (4.47) die vereinfachte Suchroutine.

```
function loc(x: keytype; t: ref): ref;                    (4.47)
begin s↑.key := x;  {Marke}
   while t↑.key <> x do
      if x< t↑.key then t := t↑.left else t := t↑.right;
   loc := t
end
```

Man beachte, dass in diesem Fall loc(x,t) den Wert s erhält, d.h. den Zeiger zur Marke, wenn kein Schlüssel mit Wert x im Baum mit Wurzel t gefunden wurde. s übernimmt einfach die Rolle des Zeigers nil.

4.4.3. Durchsuchen eines Baumes und Einfügen in einen Baum

Die Möglichkeiten der dynamischen Zuweisungstechnik mit Zugriff über Zeiger kommen in diesen Beispielen, in denen eine gegebene Menge von Daten aufgebaut wird und dann unverändert bleibt, nicht voll zum Ausdruck. Anwendungen, in denen die Struktur des Baumes selbst variiert, d.h. während der Ausführung des Programms wächst und/oder schrumpft, sind geeignetere Beispiele. An diesem Fall scheitern auch andere Darstellungen der Daten, etwa der Array, und der Baum mit durch Zeiger verknüpften Elementen erweist sich als die optimale Lösung.

Wir wollen zuerst nur den Fall des stetig wachsenden, aber niemals schrumpfenden Baumes betrachten. Ein geeignetes Beispiel ist das Erstellen eines Häufigkeitsindex, das schon im Zusammenhang mit verketteten Listen untersucht wurde und das nun nochmals aufgenommen wird. Bei diesem Problem ist eine Folge von Worten gegeben, und es ist festzustellen, wie oft jedes Wort vorkommt. Das bedeutet, dass man - ausgehend vom leeren Baum - jedes Wort in dem Baum sucht. Wird es gefunden, so wird sein Häufigkeitszähler erhöht, sonst wird es als neues Wort eingefügt (mit einem auf 1 initialisierten Zähler). Wir nennen die zugrunde liegende Aufgabe Durchsuchen des Baumes mit Einfügen. Wir gehen von folgenden Definitionen der Datentypen aus, in denen als Schlüsselwerte ganze Zahlen anstelle von Wörtern verwendet werden.

```
type ref = ↑word;                                          (4.48)
     word = record
               key: integer;
               count: integer;
               left, right: ref
            end
```

Nehmen wir ausserdem ein Quellen-File f von Schlüsseln und eine die Wurzel des Suchbaumes bezeichnende Variable root an, so können wir das Programm formulieren:

```
reset(f);                                                  (4.49)
while not eof(f) do
   begin read(f,x); search(x,root) end
```

Der Suchpfad ergibt sich direkt. Führt er jedoch in eine "Sackgasse" (d.h. zu einem leeren, durch den Zeigerwert nil bezeichneten Teilbaum), so muss das gegebene Wort an die Stelle des leeren Teilbaumes in den Baum eingefügt werden. Für den binären Baum in Fig. 4.26 und das Einfügen des Wortes "PAUL" ergibt sich das gestrichelt eingezeichnete Resultat.

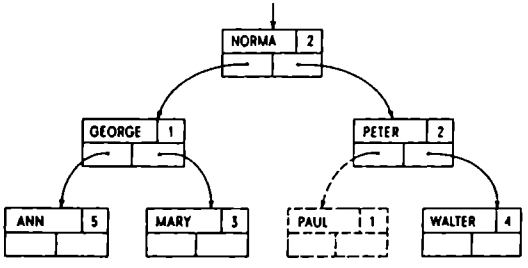


Fig. 4.26: Einfügen im geordneten binären Baum

Der ganze Vorgang wird in Programm 4.4 beschrieben. Der Suchprozess ist als rekursive Prozedur search formuliert. Man beachte, dass ihr Parameter p als Variablen-Parameter und nicht als Value-Parameter übergeben wird. Dies ist wesentlich, da im Fall des Einfügens der Variablen, die zuvor den Wert nil hatte, ein neuer Zeigerwert zuzuweisen ist. Verwendet man die Eingabe-Sequenz der bei Programm

4.3 zur Erzeugung des Baumes von Fig. 4.23 benutzten 21 Zahlen, so liefert Programm 4.4 den in Fig. 4.27 dargestellten binären Suchbaum.

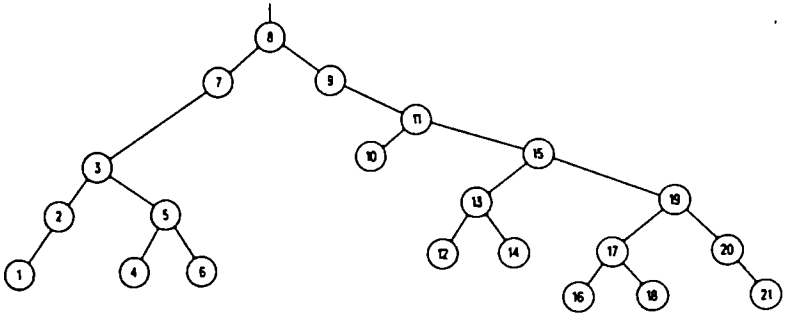


Fig. 4.27: Durch Programm 4.4 erzeugter Baum

```
program treesearch(input,output);
{binäres Durchsuchen eines Baumes mit Einfügen}
type ref = ↑word;
   word = record key: integer;
              count: integer;
              left, right: ref
          end ;
var root: ref; k: integer;

procedure printtree(w: ref; l: integer);
   var i: integer;
begin if w <> nil then
      with w↑ do
      begin printtree(left, l+1);
            for i := 1 to l do write('     ');
            writeln(key);
            printtree(right, l+1)
      end
end ;

procedure search(x: integer; var p: ref);
begin
   if p = nil then
```

```
      begin {Wort ist nicht im Baum; füge es ein}
         new(p);
         with p↑ do
            begin key := x; count := 1; left := nil; right := nil
            end
      end else
      if x < p↑.key then search(x, p↑.left) else
      if x > p↑.key then search(x, p↑.right) else
         p↑.count := p↑.count + 1
   end {search} ;

   begin root := nil; read(k);
      while k <> 0 do
         begin search(k, root); read(k)
         end ;
      printtree(root,0)
   end .
```

Programm 4.4 Durchsuchen eines Baumes mit Einfügen

Die Verwendung einer Marke vereinfacht die Aufgabe wiederum etwas (vgl. (4.50)). Selbstverständlich muss zu Beginn des Programms die Variable root durch den Zeiger auf die Marke anstatt durch den Wert nil initialisiert werden. Vor jeder Suche ist der zu suchende Wert x dem Schlüsselfeld der Marke zuzuweisen.

```
   procedure search(x: integer; var p: ref);                    (4.50)
   begin
      if x < p↑.key then search(x,p↑.left) else
      if x > p↑.key then search(x,p↑.right) else
      if p <> s then p↑.count := p↑.count + 1 else
         begin {füge ein} new (p);
            with p↑ do
            begin key := x; left := s; right := s; count := 1
            end
         end
   end
```

Noch ein letztes Mal wollen wir eine Alternative zu diesem Programm ohne Verwendung von Rekursion entwickeln. Das Vermeiden von Rekur-

sion ist nun nicht so einfach wie beim Durchsuchen ohne Einfügen, denn zum Ausführen eines Einschubs muss man den durchlaufenen Weg zumindest einen Schritt zurück kennen. Die Funktion des Erinnerns war in Programm 4.4 automatisch durch die Verwendung eines Variablen-Parameters gegeben.

Um die eingeschobene Komponente korrekt zu verknüpfen, müssen wir die Referenz zu ihrem Vorgänger kennen, und wir müssen wissen, ob sie linker oder rechter Teilbaum werden soll. Dazu werden zwei Variablen p2 und d eingeführt.

```
procedure search(x: integer);                                    (4.51)
   var p1,p2: ref; d: integer;
begin p2 := root; p1 := p2↑.right; d := 1;
   while (p1<>nil) and (d<>0) do
   begin p2 := p1;
      if x < p1↑.key then
         begin p1 := p1↑.left; d := -1 end else
      if x > p1↑.key then
         begin p1 := p1↑.right; d := 1 end else
      d := 0
   end ;
   if d = 0 then p1↑.count := p1↑.count + 1 else
      begin {Einfügen} new(p1);
         with p1↑ do
         begin key := x; left := nil; right := nil;
            count := 1
         end ;
         if d < 0 then p2↑.left := p1 else p2↑.right := p1
      end
end
```

Wie im Fall des Durchsuchens der Liste mit Einfügen werden zwei Zeiger p1 und p2 verwendet, die den Suchpfad so durchlaufen, dass p2 stets den Vorgänger von p1↑ bezeichnet. Damit zu Beginn dieses Suchprozesses diese Bedingung erfüllt ist, wird ein leeres Hilfselement eingeführt, das durch den Zeiger root bezeichnet wird. Der Anfang des wirklichen Suchbaumes wird durch den Zeiger root↑.right bezeichnet. Das Programm muss deshalb mit den Anweisungen

```
new(root); root↑.right := nil
```

anstelle der ursprünglichen Anweisung

```
root := nil
```

beginnen.

Obwohl die Suche der Hauptzweck dieses Algorithmus ist, kann er auch zum Sortieren verwendet werden. Tatsächlich hat er starke Aehnlichkeit mit der Methode des Sortierens durch Einfügen. Wegen der Verwendung einer Baumstruktur anstelle eines Array entfällt die Notwendigkeit der Verschiebung der Komponenten oberhalb der Einschubstelle. Das Sortieren mit Bäumen kann fast genauso effizient programmiert werden wie die besten bekannten Sortiermethoden mit Arrays. Allerdings sind dazu einige Vorkehrungen zu treffen. Das Finden eines gleichen Schlüssels ist nun natürlich anders zu behandeln. Wird der Fall x = p↑.key wie der Fall x > p↑.key behandelt, so stellt der Algorithmus eine stabile Sortiermethode dar, d.h. Elemente mit gleichen Schlüsseln erscheinen beim normalen Durchlaufen des Baumes in der gleichen Reihenfolge, in der sie eingefügt wurden.

Im allgemeinen gibt es bessere Wege des Sortierens, aber in Anwendungen, in denen Suchen und Sortieren gemeinsam benötigt werden, ist der Algorithmus des Durchsuchens und Einfügens mit Bäumen sehr zu empfehlen. Er wird tatsächlich sehr oft angewendet in Compilern und Datenbanken zur Organisation der Objekte, die zu speichern und wieder zuzugreifen sind. Ein gutes Beispiel hierfür ist die Konstruktion einer Cross-Reference-Liste für einen gegebenen Text. Wir wollen dieses Problem im einzelnen verfolgen.

Wir stellen uns die Aufgabe, ein Programm zu konstruieren, das (während des Lesens eines Textes f und des Druckens mit fortlaufenden Zeilennummern) alle Worte dieses Textes sammelt und dabei die Nummern der Zeilen festhält, in denen jedes Wort vorkommt. Nach Beendigung dieses Suchens soll eine Tabelle erstellt werden, die alle gesammelten Worte in alphabetischer Reihenfolge und die Listen ihres Auftretens enthält.

Offensichtlich ist der Suchbaum (auch lexikographischer Baum genannt) ein äusserst geeigneter Kandidat für die Darstellung der im Text vorkommenden Worte. Jeder Knoten enthält nun nicht nur ein Wort als Schlüsselwert, sondern ist auch Kopf einer Liste von

Zeilennummern. Wir halten jedes Auftreten in einer Variablen vom Typ item fest. Somit haben wir in diesem Beispiel beides, Bäume und lineare Listen. Das Programm 4.5 besteht aus zwei Hauptteilen, nämlich der Phase des Durchsuchens und der Phase des Druckens der Tabelle. Letztere ist eine direkte Anwendung der Routine zum Durchlaufen eines Baumes, wobei der Besuch jedes Knotens das Drucken des Schlüsselwertes (word) und das Durchsuchen der zugeordneten Liste von Zeilennummern (items) zur Folge hat. Hier folgen weitere Erläuterungen für Programm 4.5 zum Erstellen einer Cross-Reference-Liste.

1. Ein Wort wird als eine Folge von Buchstaben und Ziffern definiert, die mit einem Buchstaben beginnt.
2. Nur die ersten c1 Zeichen werden als Schlüssel aufbewahrt. Somit werden zwei Worte, die sich in ihren ersten c1 Zeichen nicht unterscheiden, als identisch angesehen.
3. Die c1 Zeichen werden in einen Array id (vom Typ alfa) gepackt. Für hinreichend kleines c1 können viele Computer solche gepackten Arrays durch eine einzige Instruktion vergleichen.
4. Die Variable k1 wird als Index zur Aufrechterhaltung der folgenden invarianten Bedingung für den Zeichenpuffer a verwendet:
   a[i] = ' ' für i = k1+1...c1 .
   Worte mit weniger als c1 Zeichen werden durch eine entsprechende Zahl von Leerzeichen erweitert.
5. Vorzugsweise sollen die Zeilennummern in aufsteigender Reihenfolge in der Cross-Reference-Tabelle gedruckt werden. Aus diesem Grund sind die Listen der Elemente in der gleichen Reihenfolge aufzubauen, in der sie zum Drucken durchlaufen werden. Diese Anforderung legt die Verwendung zweier Zeiger in jedem Wortknoten nahe, einen zur Angabe des ersten und einen zur Angabe des letzten Elementes der Liste.
6. Der Prozess des Durchsuchens ist so aufgebaut, dass Worte in Anführungszeichen und Kommentaren nicht in die Tabelle aufgenommen werden unter der Voraussetzung, dass Zitate und Kommentare nicht über das Zeilenende hinausgehen.

Tabelle 4.4 zeigt das Resultat der Anwendung von Programm 4.5 auf einen kurzen Programmtext.

```
program crossref(f,output);
{Erstellen einer Cross-Reference-Tabelle mittels binärem Baum}
const c1 = 10;   {Wortlänge}
      c2 =  8;   {Zahlen pro Zeile}
      c3 =  6;   {Ziffern pro Zahl}
      c4 = 9999; {max. Zeilennummer}
type  alfa = packed array [1..c1] of char;
      wordref = ↑word;  itemref = ↑item;
      word = record key: alfa;
                    first, last: itemref;
                    left, right: wordref
             end ;
      item = packed record
                    lno: 0..c4;
                    next: itemref
             end ;
var   root: wordref;
      k,k1: integer;
      n: integer;          {momentane Zeilennummer}
      id: alfa;
      f: text;
      a: array [1..c1] of char;

procedure search(var w1: wordref);
   var w: wordref; x: itemref;
begin w := w1;
   if w = nil then
   begin new(w); new(x);
      with w↑ do
      begin key := id; left := nil; right := nil;
         first := x; last := x
      end ;
      x↑.lno := n; x↑.next := nil; w1 := w
   end else
   if id < w↑.key then search(w↑.left) else
   if id > w↑.key then search(w↑.right) else
   begin new(x); x↑.lno := n; x↑.next := nil;
      w↑.last↑.next := x; w↑.last := x
   end
end {search} ;
```

```
procedure printtree(w: wordref);

   procedure printword(w: word);
      var l: integer; x: itemref;
   begin write(' ', w.key);
      x := w.first; l := 0;
      repeat if l = c2 then
             begin writeln;
                l := 0; write(' ':c1+1)
             end ;
         l := l+1; write(x↑.lno:c3); x := x↑.next
      until x = nil;
      writeln
   end {printword} ;

begin if w <> nil then
   begin printtree(w↑.left);
         printword(w↑); printtree(w↑.right)
   end
end {printtree} ;

begin root := nil; n := 0; k1 := c1;
   page(output); reset(f);
   while not eof(f) do
   begin if n = c4 then n := 0;
      n := n+1; write(n:c3);       {nächste Zeile}
      write(' ');
      while not eoln(f) do
      begin {durchsuche die nichtleere Zeile}
         if f↑ in ['a'..'z'] then
         begin k := 0;
            repeat if k < c1 then
               begin k := k+1;  a[k] := f↑;
               end ;
               write(f↑); get(f)
            until not (f↑ in ['a'..'z','0'..'9']);
            if k >= k1 then k1 := k else
            repeat a[k1] := ' ';  k1 := k1-1
            until k1 = k;
            pack(a,1,id); search(root)
```

```
         end else
         begin {teste auf Anführungszeichen oder Kommentar}
            if f↑ = '''' then
               repeat write(f↑); get(f)
               until f↑ = '''' else
            if f↑ = '{' then
               repeat write(f↑); get(f)
               until f↑ = '}' ;
            write(f↑); get(f)
         end
      end ;
      writeln; get(f)
   end ;

   page(output); printtree(root);
end .
```

Programm 4.5

```
 1 PROGRAM PERMUTE(OUTPUT);
 2    CONST N = 4;
 3    VAR I: INTEGER;
 4        A: ARRAY [1..N] OF INTEGER;
 5
 6    PROCEDURE PRINT;
 7       VAR I: INTEGER;
 8    BEGIN FOR I := 1 TO N DO WRITE(A[I]:3);
 9       WRITELN
10    END {PRINT} ;
11
12    PROCEDURE PERM(K: INTEGER);
13       VAR I,X: INTEGER;
14    BEGIN
15       IF K = 1 THEN PRINT ELSE
16       BEGIN PERM(K-1);
17          FOR I := 1 TO K-1 DO
18          BEGIN X := A[I]; A[I] := A[K]; A[K]:= X;
19             PERM(K-1);
20             X := A[I]; A[I] := A[K]; A[K] := X
```

```
 21          END
 22       END
 23    END {PERM} ;
 24
 25 BEGIN
 26    FOR I := 1 TO N DO A[I] := I;
 27    PERM(N)
 28 END .
```

ARRAY	4							
A	4	8	18	18	18	18	20	20
	20	20	26					
BEGIN	8	14	16	18	25			
CONST	2							
DO	8	17	26					
ELSE	15							
END	10	21	22	23	28			
FOR	8	17	26					
IF	15							
INTEGER	3	4	7	12	13			
I	3	7	8	8	13	17	18	18
	20	20	26	26	26			
K	12	15	16	17	18	18	19	20
	20							
N	2	4	8	26	27			
OF	4							
OUTPUT	1							
PERMUTE	1							
PERM	12	16	19	27				
PRINT	6	15						
PROCEDURE	6	12						
PROGRAM	1							
THEN	15							
TO	8	17	26					
VAR	3	7	13					
WRITELN	9							
WRITE	8							
X	13	18	18	20	20			

Tabelle 4.4

4.4.4. Löschen in Bäumen

Wir wenden uns nun dem dem Einfügen entgegengesetzten Problem zu, nämlich dem Löschen (deletion). Unsere Aufgabe ist es, einen Algorithmus für das Löschen aufzustellen, d.h. einen Knoten mit Schlüssel x aus einem Baum mit geordneten Schlüsseln zu entfernen. Leider ist das Entfernen eines Elementes im allgemeinen nicht so einfach wie das Einfügen. Es kann direkt ausgeführt werden, wenn das zu löschende Element ein Endknoten oder ein Knoten mit einem einzigen Nachfolger ist. Die Schwierigkeit liegt im Entfernen eines Elementes mit zwei Nachfolgern, da wir mit einem Zeiger nicht in zwei Richtungen zeigen können. In diesem Fall muss das entfernte Element entweder durch das grösste Element des linken Teilbaumes oder durch das kleinste Element des rechten Teilbaumes ersetzt werden. Beide haben höchstens einen Nachfolger. Die rekursive Prozedur delete in (4.52) zeigt die Einzelheiten und unterscheidet zwischen drei Fällen:

1. Es gibt keine Komponente mit einem Schlüssel gleich x.
2. Die Komponente mit Schlüssel x hat höchstens einen Nachfolger.
3. Die Komponente mit Schlüssel x hat zwei Nachfolger.

```
procedure delete(x: integer; var p: ref):                    (4.52)
  var q: ref;
    procedure del(var r: ref);
    begin if r↑.right <> nil then del(r↑.right) else
          begin p↑.key := r↑.key; q↑.count := r↑.count;
             q := r; r := r↑.left
          end
    end;
begin {delete}
   if p = nil then writeln(' Wort ist nicht im Baum') else
   if x < p↑.key then delete(x,p↑.left) else
   if x > p↑.key then delete(x,p↑.right) else
   begin {entferne p↑} q := p;
      if q↑.right = nil then p := q↑.left else
      if q↑.left  = nil then p := q↑.right
                        else del(q↑.left);
      {dispose(q)}
   end
end {delete}
```

Die rekursive Hilfs-Prozedur del wird nur im 3. Fall aufgerufen. Sie geht auf dem rechtesten Zweig des linken Teilbaumes des zu löschenden Elementes q↑ "hinab" und ersetzt die wesentliche Information (Schlüssel und Zähler) in q↑ durch die entsprechenden Werte der rechtesten Komponente r↑ dieses linken Teilbaumes, wonach dann r↑ weggenommen werden kann. Die nicht weiter ausgeführte Prozedur dispose(q) kann als die inverse oder entgegengesetzte von new(q) betrachtet werden. Während letztere Speicher für eine neue Komponente zuweist, kann erstere verwendet werden, um dem System mitzuteilen, dass der durch q↑ besetzte Speicher zur weiteren Verwendung wieder zur Verfügung steht (eine Art Kreislauf des Speicherplatzes).

Zur Illustration des Ablaufs der Prozedur (4.52) beziehen wir uns auf Fig. 4.28. Man gehe aus von Baum (a) und entferne nacheinander die Knoten mit den Schlüsseln 13, 15, 5 und 10. Die sich ergebenden Bäume zeigen Fig. 4.28 b-e.

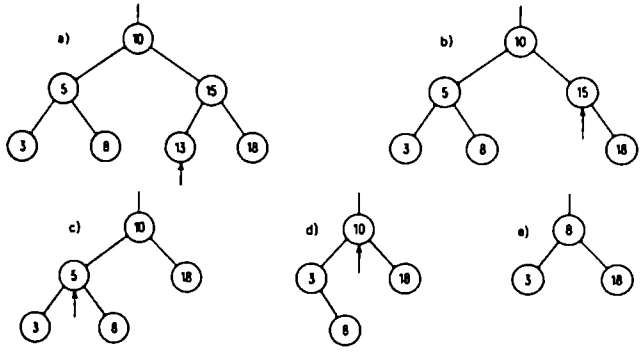


Fig. 4.28: Löschen von Elementen im geordneten Baum

4.4.5. Analyse des Durchsuchens eines Baumes und Einfügens in einen Baum

Es ist nur natürlich, wenn man bezüglich des eben dargestellten Algorithmus zum Durchsuchen und Einfügen bei Bäumen misstrauisch ist. Zumindest sollte man skeptisch bleiben, bis weitere Einzelheiten über sein Verhalten erörtert worden sind. Viele Programmierer

sind besorgt über die eigenartige Tatsache, dass sie im allgemeinen nicht wissen, wie der Baum wächst; wir haben keine Ahnung, welche Form er annehmen wird. Wir können nur vermuten, dass er höchstwahrscheinlich nicht ganz ausgeglichen sein wird. Die mittlere Zahl der zum Finden eines Schlüssels in einem völlig ausgeglichenen Baum mit n Knoten benötigten Vergleiche beträgt ungefähr h = log n. Daher wird die Zahl der Vergleiche bei einem durch diesen Algorithmus generierten Baum grösser sein als h. Aber um wieviel?

Es ist zunächst einfach, den schlimmsten Fall zu finden. Nehmen wir an, dass alle Schlüssel bereits in streng aufsteigender (oder absteigender) Reihenfolge auftreten. Dann wird jeder Schlüssel sofort rechts (links) an seinen Vorgänger angehängt, und der resultierende Baum ist vollständig degeneriert, d.h. er wird zu einer linearen Liste. Der mittlere Suchaufwand ist dann n/2 Vergleiche. Dieser schlimmste Fall führt natürlich zu einer sehr schlechten Leistung des Such-Algorithmus und scheint unsere Skepsis völlig zu rechtfertigen. Es bleibt natürlich noch die Frage, wie wahrscheinlich dieser Fall ist. Genauer gesagt, sind wir an der Länge a[n] des Suchpfades interessiert, gemittelt über alle n Schlüssel und über alle n! Bäume, die sich aus den n! Permutationen der ursprünglichen n verschiedenen Schlüssel ergeben. Das Problem der algorithmischen Analyse wird hier sehr deutlich. Sie sei hier ausgeführt sowohl wegen ihres Wertes als typisches Beispiel der Analyse von Algorithmen wie auch wegen der praktischen Bedeutung des Resultats.

Gegeben seien n verschiedene Schlüssel mit den Werten 1, 2, ..., n in zufälliger Reihenfolge. Die Wahrscheinlichkeit, dass der erste Schlüssel - der übrigens Wurzelknoten wird - den Wert i hat, ist 1/n. Sein linker Teilbaum wird zuletzt i-1 Knoten enthalten und sein rechter Teilbaum n-i Knoten (vgl. Fig. 4.29).

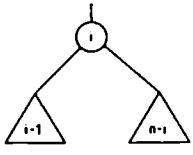


Fig. 4.29: Zur Bestimmung der mittleren Weglänge

Die mittlere Weglänge im linken Teilbaum wird durch a[i-1] bezeichnet, die im rechten mit a[n-i], wieder unter der Voraussetzung, dass alle möglichen Permutationen der restlichen n-1 Schlüssel gleich wahrscheinlich sind. Die mittlere Weglänge in einem Baum mit n Knoten ist die Summe der Produkte von der Stufe jedes Knotens mit seiner Wahrscheinlichkeit des Zugriffs. Werden alle Knoten mit gleicher Wahrscheinlichkeit benötigt, so gilt

$$a[n] = (1/n) * \sum_{i=1}^{n} p[i] \qquad (4.53)$$

Im Baum von Abb. 4.29 teilen wir die Knoten in drei Klassen ein:

1. Die i-1 Knoten im linken Teilbaum haben eine mittlere Weglänge a[i-1]+1.

2. Die Wurzel hat Weglänge 1.

3. Die n-i Knoten im rechten Teilbaum haben eine mittlere Weglänge a[n-i]+1 .

Somit kann (4.53) als Summe der drei Ausdrücke geschrieben werden

$$a[n](i) = (a[i-1]+1)*(i-1)/n + 1/n + (a[n-i]+1)*(n-i)/n \qquad (4.54)$$

Die gesuchte Grösse a[n] wird nun abgeleitet als Mittel der a[n](i) über alle i = 1...n, d.h. über alle Bäume mit dem Schlüssel 1, 2, ... , n in der Wurzel.

$$a[n] = (1/n) * \sum_{i=1}^{n} [(a[i-1]+1)*(i-1)/n + 1/n + (a[n-i]+1)*(n-i)/n]$$

$$= 1 + (1/n**2) * \sum_{i=1}^{n} [(i-1)a[i-1] + (n-i)*a[n-i]]$$

$$= 1 + (2/n^{**}2) * \sum_{i=1}^{n} (i-1)*a[i-1]$$

$$= 1 + (2/n^{**}2) * \sum_{i=1}^{n-1} i*a[i] \qquad (4.55)$$

Die Gleichung (4.55) ist eine Rekursions-Relation für a[n] von der Form a[n] = f(a[1],a[2],...,a[n-1]). Daraus können wir auf folgende Art eine einfachere Rekursions-Relation der Form a[n] = g(a[n-1]) herleiten:

$$(1) \quad a[n] = 1 + (2/n^{**}2) * \sum_{i=1}^{n-1} i*a[i]$$

$$= 1 + (2/n^{**}2)*(n-1)*a[n-1] + (2/n^{**}2) * \sum_{i=1}^{n-2} i*a[i]$$

$$(2) \quad a[n-1] = 1 + 2/(n-1)^{**}2 * \sum_{i=1}^{n-2} i*a[i]$$

Multiplikation von (2) mit ((n-1)/n)**2 ergibt

$$(3) \quad 2/n^{**}2 * \sum_{i=1}^{n-2} i*a[i] = ((n-1)/n)^{**}2 * (a[n-1] - 1)$$

Einsetzen von (3) in (1) führt zu

$$a[n] = 1/n^{**}2 * ((n^{**}2-1)*a[n-1] + 2*n - 1) \qquad (4.56)$$

Es stellt sich heraus, dass a[n] in nicht rekursiver geschlossener Form mit der harmonischen Funktion H[n] dargestellt werden kann:

$$H[n] = 1 + 1/2 + 1/3 + \ldots + 1/n$$

$$a[n] = 2*(n+1)/n * H[n] - 3 \qquad (4.57)$$

(Der kritische Leser sollte nachprüfen, dass (4.57) die Rekursions-Relation (4.56) erfüllt.)

Aus Eulers Formel (mit der Eulerschen Konstanten g ≈ 0.577)

$$H[n] = g + \ln(n) + 1/(12*n**2) + \ldots$$

leiten wir für grosse n folgende Relation ab:

$$a[n] \approx 2*(\ln(n) + g) - 3 = 2*\ln(n) - c \ .$$

Da die mittlere Weglänge im völlig ausgeglichenen Baum ungefähr

$$a'[n] = \log(n) - 1 \qquad (4.58)$$

ist, erhalten wir unter Vernachlässigung der für grosse n unbedeutenden konstanten Glieder:

$$\lim_{n\to\infty} a[n]/a'[n] = \lim_{n\to\infty} 2*\ln(n)/\log(n) = 2*\ln(2) \approx 1.386 \qquad (4.59)$$

Was bedeutet nun das Resultat (4.59) dieser Analyse? Es sagt uns, dass wir bei dauernder Konstruktion eines völlig ausgeglichenen Baumes anstelle des durch Programm 4.4 entstehenden zufälligen Baumes - immer unter der Voraussetzung, dass alle Schlüssel mit gleicher Wahrscheinlichkeit besucht werden - eine mittlere Verbesserung der Länge des Suchpfades um höchstens 39% erwarten können. Die Betonung liegt auf dem Wort "mittlere", denn die Verbesserung kann in Wirklichkeit im ungünstigsten Fall sehr viel grösser sein, wenn nämlich der generierte Baum völlig zu einer Liste degeneriert, was allerdings sehr unwahrscheinlich ist (wenn alle Permutationen der n einzufügenden Schlüssel gleich wahrscheinlich sind). In diesem Zusammenhang ist es bemerkenswert, dass die zu erwartende mittlere Weglänge des "zufälligen" Baumes auch streng logarithmisch mit der Zahl seiner Knoten zunimmt, obwohl für den schlimmsten Fall die Weglänge linear wächst.

Der Wert 39% bildet eine Grenze für die zusätzlichen Anstrengungen, die man sinnvollerweise für irgendeine Art der Neuorganisation der Baumstruktur beim Einfügen von Elementen aufwendet. Natürlich beeinflusst das Verhältnis r zwischen der Häufigkeit des Zugriffs

und des Einfügens von Knoten die für ein solches Vorhaben rentable Grenzen beträchtlich. Je grösser dieses Verhältnis ist, um so komplexer kann die Prozedur zur Neuorganisation sein. Der Wert 39% ist klein genug, dass sich bei den meisten Anwendungen Verbesserungen des Algorithmus für das direkte Einfügen in den Baum nicht lohnen, wenn nicht die Zahl der Knoten und das Verhältnis von Zugriff zu Einfügen gross sind.

4.4.6. Ausgeglichene Bäume

Aus der vorangehenden Diskussion geht hervor, dass eine Prozedur zum Einfügen, die jedesmal die völlig ausgeglichene Struktur des Baumes wiederherstellt, kaum rentabel sein kann, da die Wiederherstellung der völligen Ausgeglichenheit nach einem willkürlichen Einfügen eine sehr knifflige Operation ist. Verbesserungen werden durch eine schwächere Definition der "Ausgeglichenheit" möglich. Solch ein Kriterium für "unvollständige" Ausgeglichenheit sollte einfachere Prozeduren der Neuorganisation ergeben auf Kosten einer nur geringen Verschlechterung der mittleren Suchleistung.

Eine solche Definition der Ausgeglichenheit wurde von Adelson-Velskii und Landis [1] gegeben:

> Ein Baum ist genau dann ausgeglichen, wenn sich für jeden Knoten die Höhen der Teilbäume um höchstens 1 unterscheiden.

Bäume, die diese Bedingung erfüllen, heissen nach ihren Schöpfern oft AVL-Bäume. Wir werden sie einfach ausgeglichene Bäume nennen, da dieses Kriterium der Ausgeglichenheit als das geeignetste erscheint. (Man beachte, dass alle völlig ausgeglichenen Bäume auch AVL-ausgeglichen sind.)

Diese Definition ist nicht nur einfach, sondern führt auch zu einer sehr handlichen Prozedur zum Ausgleichen und zu einem mittleren Suchpfad, dessen Länge praktisch identisch ist mit der eines völlig ausgeglichenen Baumes.

Die folgenden Operationen können auf ausgeglichenen Bäumen selbst im schlimmsten Fall in O(log n) Zeiteinheiten ausgeführt werden.

1. Suchen eines Knotens mit gegebenem Schlüssel.
2. Einfügen eines Knotens mit gegebenem Schlüssel.
3. Löschen eines Knotens mit gegebenem Schlüssel.

Diese Aussagen sind direkte Konsequenzen eines von Adelson-Velskii und Landis bewiesenen Theorems, das garantiert, dass ein ausgeglichener Baum nie mehr als 45% höher ist als sein völlig ausgeglichenes Gegenstück, unabhängig von der Anzahl der vorhandenen Knoten. Bezeichnen wir die Höhe eines ausgeglichenen Baumes mit n Knoten durch hb(n), so gilt

$$\log(n+1) <= hb(n) <= 1.4404*\log(n+2) - 0.328 \qquad (4.60)$$

Das Optimum wird natürlich erreicht für einen völlig ausgeglichenen Baum mit n = 2**k-1. Welche Struktur hat aber der schlechtest AVL-ausgeglichene Baum?

Um die maximale Höhe h aller ausgeglichenen Bäume mit n Knoten zu finden, wollen wir von einer festen Höhe h ausgehen und versuchen, den ausgeglichenen Baum mit minimaler Anzahl von Knoten zu konstruieren. Diese Strategie ist zu empfehlen, da wie im Fall minimaler Höhe h der Wert nur für bestimmte Werte von n angenommen wird. Diesen Baum mit der Höhe h bezeichnen wir mit T[h]. Natürlich ist T[0] der leere Baum und T[1] der Baum mit einem einzigen Knoten. Um den Baum T[h] mit h > 1 zu konstruieren, wollen wir die Wurzel mit zwei Teilbäumen versehen, die wiederum eine minimale Anzahl von Knoten haben. Die Teilbäume sind also wieder T's. Natürlich muss ein Teilbaum die Höhe h-1 haben, der andere kann dann eine um 1 kleinere Höhe, nämlich h-2 haben. Fig. 4.30 zeigt die Bäume mit den Höhen 2, 3 und 4. Da das Prinzip ihrer Zusammensetzung sehr stark dem der Fibonacci-Zahlen gleicht, heissen sie Fibonacci-Bäume. Sie sind folgendermassen definiert:

1. Der leere Baum ist ein Fibonacci-Baum der Höhe 0.
2. Ein einzelner Knoten ist Fibonacci-Baum der Höhe 1.
3. Sind T[h-1] und T[h-2] Fibonacci-Bäume mit den Höhen h-1 und h-2, so ist T[h] = <T[h-1], x, T[h-2]> ein Fibonacci-Baum der Höhe h.
4. Keine anderen Bäume sind Fibonacci-Bäume.

Die Zahl der Knoten von T[h] ist durch folgende einfache Rekursions-Relation definiert:

$$N[0] = 0, \qquad N[1] = 1$$
$$N[h] = N[h-1] + 1 + N[h-2] \qquad (4.61)$$

Die N[i] sind die Zahlen von Knoten, für die der schlimmste Fall (die obere Grenze von h) in (4.60) erreicht werden kann.

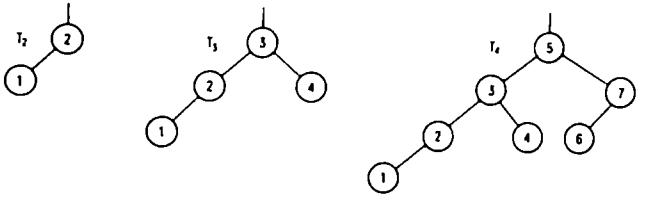


Fig. 4.30: Fibonacci-Bäume der Höhen 2,3 und 4

4.4.7. Einfügen in ausgeglichene Bäume

Wir wollen nun betrachten, was geschieht, wenn ein neuer Knoten in einen ausgeglichenen Baum eingefügt wird. Gegeben sei eine Wurzel r mit den linken und rechten Teilbäumen L und R, dann sind drei Fälle zu unterscheiden. Der neue Knoten sei in L einzufügen:

(a) hL = hR : Die Höhen von L und R werden verschieden, aber das Kriterium der Ausgeglichenheit wird nicht verletzt.

(b) hL < hR : Die Höhen von L und R werden gleich, d.h. die Ausgeglichenheit wird sogar besser.

(c) hL > hR : Die Ausgeglichenheit wird zerstört, und der Baum muss umstrukturiert werden.

Im Baum von Fig. 4.31 können Knoten mit Schlüsseln 9 und 11 ohne zusätzliches Ausgleichen eingefügt werden; der Baum mit Wurzel 10 wird einseitig (Fall (a)), der mit Wurzel 8 besser ausgeglichen (Fall (b)). Einfügen von Knoten 1, 3, 5 oder 7 erfordert jedoch anschliessendes Ausgleichen.

Eine sorgfältige Ueberprüfung der Situation ergibt, dass nur zwei wesentlich verschiedene Konstellationen individuell zu behandeln sind. Die restlichen können durch Symmetrie-Ueberlegungen auf diese beiden zurückgeführt werden. Fall 1 ist durch Einfügen der Schlüssel 1 oder 3 in den Baum von Abb. 4.31 charakterisiert, Fall 2 durch Einfügen der Knoten 5 oder 7.

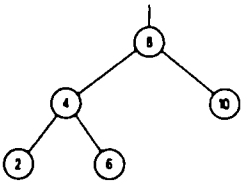


Fig. 4.31: Ausgewogener Baum (AVL)

Beide Fälle werden in Fig. 4.32 verallgemeinert, wobei Rechtecke die Teilbäume darstellen und die durch Einfügen hinzugekommene Höhe durch Kreuze markiert ist.

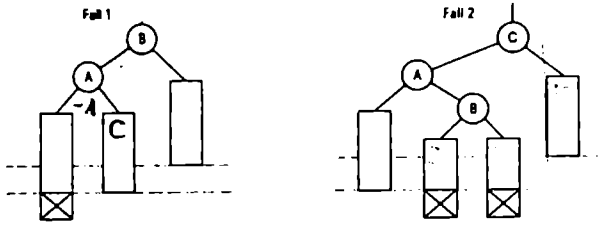


Fig. 4.32: Nach Einfügen eines Elementes nicht mehr ausgewogene Bäume

Einfache Umformungen der beiden Strukturen stellen die gewünschte Ausgeglichenheit wieder her. Das Ergebnis ist in Fig. 4.33 gezeigt: man beachte, dass nur Bewegungen in vertikaler Richtung erlaubt sind, während die relativen horizontalen Positionen der gezeigten Knoten und Teilbäume unverändert bleiben müssen.

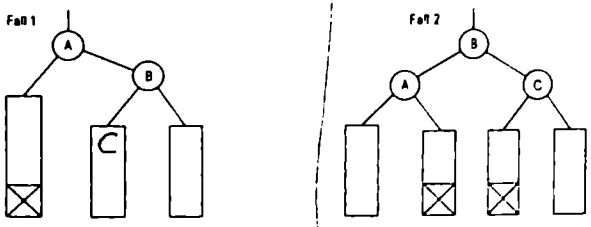


Fig. 4.33: Durch Ausgleichen wiederhergestelltes Gleichgewicht der Bäume von Fig. 4.32

Ein Algorithmus zum Einfügen und Ausgleichen hängt entscheidend von der Art der Speicherung der Information ab, die die Ausgeglichenheit des Baumes angibt. Bei einer extremen Lösung liegt die Information der Ausgeglichenheit implizit in der Baumstruktur selbst. In diesem Fall aber muss der Grad der Ausgeglichenheit eines Knotens jedesmal wieder festgestellt werden, wenn der Knoten bei einem Einfügen berührt wird, woraus sich ein ausserordentlich grosser Verwaltungsaufwand ergibt. Das andere Extrem ist die Erweiterung jedes Knotens um den explizit gespeicherten Grad der Ausgeglichenheit. Die Definition (4.42) des Typs node wird dann erweitert zu

```
type node = record ...                                    (4.62)
              left, right: ref;
              bal: -1 .. +1
            end
```

Wir werden im folgenden den Grad der Ausgeglichenheit eines Knotens als Differenz der Höhen des rechten und linken Teilbaumes interpretieren. Der zu erstellende Algorithmus wird sich auf den Typ node von (4.62) beziehen.

Der Prozess des Einfügens eines Knotens besteht im wesentlichen aus drei aufeinanderfolgenden Teilen:

1. Man folge dem Suchpfad, bis feststeht, dass der Schlüssel nicht schon im Baum vorhanden ist.

2. Man füge den neuen Knoten ein und bestimme den sich ergebenden

Grad der Ausgeglichenheit.

3. Man gehe dem Suchpfad entlang zurück und teste in jedem Knoten den Grad der Ausgeglichenheit.

Obwohl bei dieser Methode einige überflüssige Tests ausgeführt werden (wenn Ausgeglichenheit einmal erreicht ist, braucht sie bei den Vorgängern des Knotens nicht mehr geprüft zu werden), so wollen wir doch zunächst an diesem sicherlich richtigen Schema festhalten, da es durch reine Erweiterung der bereits in Programm 4.4 erstellten Prozedur zum Suchen und Einfügen implementiert werden kann. Diese Prozedur beschreibt nämlich die zur Suche eines jeden einzelnen Knotens notwendige Operation, und wegen ihrer rekursiven Formulierung kann sie "auf dem Rückweg entlang dem Suchpfad" leicht eine zusätzliche Operation ausführen. Bei jedem Schritt ist Information zu übergeben, die angibt, ob die Höhe des Teilbaumes (in den ein Element eingefügt wurde) zugenommen hat oder nicht. Wir erweitern daher die Parameter-Liste der Prozedur um die Boolesche Variable h, die besagt, "die Höhe des Teilbaumes hat zugenommen". Natürlich muss h ein Variablen-Parameter sein, da er ein Resultat übermittelt.

Nehmen wir nun an, dass der Prozess zu einem Knoten p↑ aus der linken Verzweigung (vgl. Fig. 4.32) mit der Meldung zurückkommt, dass die Höhe zugenommen hat. Wir müssen jetzt entsprechend den Höhen der Teilbäume vor dem Einfügen zwischen drei Fällen unterscheiden:

1. hL < hR , p↑.bal = +1, die bisherige Unausgeglichenheit bei p wurde ausgeglichen.

2. hL = hR , p↑.bal = 0, das Gewicht verlagert sich nun nach links.

3. hL > hR , p↑.bal = −1, erneutes Ausgleichen wird notwendig.

Im dritten Fall bestimmt der Grad der Ausgeglichenheit der Wurzel des linken Teilbaumes (das ist p1↑.bal), ob Fall 1 oder 2 von Fig. 4.32 vorliegt. Wenn auch der linke Teilbaum dieses Knotens höher ist als der rechte, haben wir es mit Fall 1 zu tun, sonst mit Fall 2. (Man überzeuge sich selbst, dass ein linker Teilbaum mit Grad

null der Ausgeglichenheit in seiner Wurzel in dieser Situation nicht vorkommen kann.) Die zum erneuten Ausgleichen notwendigen Operationen werden vollständig durch eine Folge von Zeigerzuweisungen ausgedrückt. Tatsächlich werden Zeiger zyklisch ausgetauscht, so dass entweder eine einfache oder doppelte Rotation der zwei oder drei beteiligten Knoten ausgeführt wird. Zusätzlich zur Rotation der Zeiger sind die entsprechenden Angaben der Ausgeglichenheit nachzuführen. Einzelheiten sind in der Prozedur (4.63) zum Suchen, Einfügen und Ausgleichen enthalten.

Fig. 4.34 zeigt das Prinzip des Vorgehens. Man betrachte den binären Baum (a), der nur aus zwei Knoten besteht. Einfügen von Schlüssel 7 führt zunächst zu einem unausgeglichenen Baum (d.h. einer linearen Liste). Das Ausgleichen bedingt eine einfache RR Rotation und ergibt den völlig ausgeglichenen Baum (b). Weiteres Einfügen von Knoten 2 und 1 führt zu einem unausgeglichenen Teilbaum mit Wurzel 4 Dieser Teilbaum wird durch eine einzige LL Rotation ausgeglichen (d). Das anschliessende Einfügen von Schlüssel 3 verletzt sofort im Wurzelknoten 5 das Kriterium der Ausgeglichenheit. Es ist in Baum (e) wieder erfüllt nach einer komplizierteren LR Doppelrotation. Knoten 5 ist der einzige Kandidat, der beim nächsten Einfügen unausgeglichen werden kann. Tatsächlich verursacht das Einfügen von Knoten 6 den vierten in (4.63) angegebenen Fall eines Ausgleichs, die RL Doppelrotation. Fig. 4.34(f) enthält die endgültige Form des Baumes.

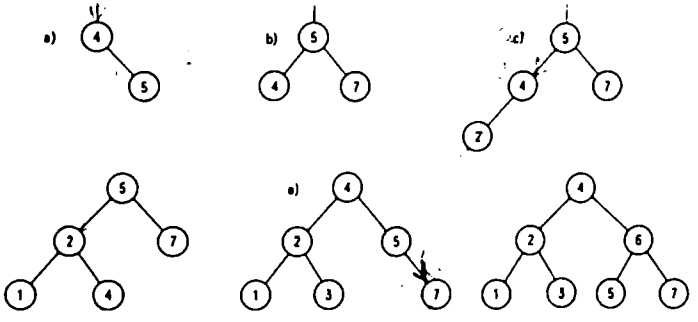


Fig. 4.34: Einfügen von Elementen im ausgewogenen Baum

```
procedure search(x: integer; var p: ref; var h: boolean);
   var p1,p2: ref;     {h = false}                          (4.63)
begin
   if p = nil then
   begin {Wort nicht im Baum; füge es ein}
      new(p); h := true;
      with p↑ do
      begin key := x; count := 1;
         left := nil; right := nil; bal := 0
      end
   end else
   if x < p↑.key then
   begin search(x, p↑.left, h);
      if h then    {linker Ast wurde grösser}
      case p↑.bal of
     1:  begin p↑.bal := 0; h := false
         end :
     0:  p↑.bal := -1;
    -1:  begin {erneutes Ausgleichen} p1 := p↑.left;
            if p1↑.bal = -1 then
            begin {einfache LL-Rotation}
               p↑.left := p1↑.right; p1↑.right := p;
               p↑.bal := 0; p := p1
            end else
            begin {doppelte LR-Rotation}  p2 := p1↑.right;
               p1↑.right := p2↑.left; p2↑.left := p1;
               p↑.left := p2↑.right; p2↑.right := p;
               if p2↑.bal = -1 then p↑.bal := +1
                               else p↑.bal := 0;
               if p2↑.bal = +1 then p1↑.bal := -1
                               else p1↑.bal := 0;
               p := p2
            end :
            p↑.bal := 0; h := false
         end
      end
   end else
   if x > p↑.key then
   begin search(x, p↑.right, h);
      if h then    {rechter Ast wurde grösser}
```

```
        case p↑.bal of
      -1:  begin p↑.bal := 0; h := false
           end ;
       0:  p↑.bal := +1;
       1:  begin {erneutes Ausgleichen} p1 := p↑.right;
             if p1↑.bal = +1 then
             begin {einfache RR-Rotation}
               p↑.right := p1↑.left; p1↑.left := p;
               p↑.bal := 0; p := p1
             end else
             begin {doppelte RL-Rotation}  p2 := p1↑.left;
               p1↑.left := p2↑.right; p2↑.right := p1;
               p↑.right := p2↑.left; p2↑.left := p;
               if p2↑.bal = +1 then p↑.bal := -1
                               else p↑.bal := 0;
               if p2↑.bal = -1 then p1↑.bal := +1
                               else p1↑.bal := 0;
               p := p2
             end ;
             p↑.bal := 0; h := false
           end
       end
    end
    else
    begin p↑.count := p↑.count + 1; h := false
    end
  end {search}
```

Zwei Fragen bezüglich der Leistung des Algorithmus zum Einfügen in einen ausgeglichenen Baum sind von besonderem Interesse.

1. Wie grosa ist die zu erwartende Höhe des erstellten ausgeglichenen Baumes, wenn alle n! Permutationen von n Schlüsseln mit gleicher Wahrscheinlichkeit vorkommen?

2. Mit welcher Wahrscheinlichkeit erfordert ein Einfügen erneutes Ausgleichen?

Die mathematische Behandlung dieses komplizierten Algorithmus ist immer noch ein offenes Problem. Empirische Tests bestätigen die

Vermutung, dass die zu erwartende Höhe des durch (4.63) generierten ausgeglichenen Baumes h = log(n)+c ist mit kleinem konstantem c (c = 0.25). Damit verhält sich der AVL-ausgeglichene Baum praktisch so gut wie der völlig ausgeglichene Baum, obwohl seine Struktur viel einfacher beizubehalten ist. Empirische Anzeichen deuten auch darauf hin, dass erneutes Ausgleichen im Mittel ungefähr einmal auf je zwei Erweiterungen nötig ist. Dabei sind einfache und doppelte Rotationen gleich wahrscheinlich. Das Beispiel in Fig. 4.34 wurde offenbar sorgfältig gewählt, um so viele Rotationen wie möglich in einer kleinen Zahl von Erweiterungen zu zeigen!

Die Komplexität der Operationen zum Ausgleich legt nahe, ausgeglichene Bäume nur dann zu verwenden, wenn der Zugriff zur Information wesentlich häufiger ist als Erweiterungen. Dies gilt ganz besonders deshalb, weil die Knoten solcher Suchbäume gewöhnlich als dicht gepackte Records implementiert werden, um Speicherplatz zu sparen. Die Geschwindigkeit des Zugriffs und des Nachführens des Grades der Ausgeglichenheit - das jeweils nur zwei Bits betrifft - ist daher oft ein entscheidender Faktor für die Effizienz der Operation des Ausgleichens. Empirische Abschätzungen zeigen, dass ausgeglichene Bäume viel von ihrer Attraktivität verlieren, wenn die Records dicht gepackt werden müssen. Es ist wirklich schwierig, den primitiven, einfachen Algorithmus für das Suchen und Einfügen in Bäume zu übertreffen!

4.4.8. Löschen in ausgeglichenen Bäumen

Unsere Erfahrung mit Löschen in Bäumen lässt vermuten, dass auch Löschen in ausgeglichenen Bäumen komplizierter ist als Einfügen. Dies stimmt tatsächlich, obwohl die Operation des Ausgleichens im wesentlichen die gleiche ist wie die des Einfügens. Insbesondere besteht das Ausgleichen entweder in einer einfachen oder doppelten Rotation der Knoten. Ausgangspunkt für das Löschen in ausgeglichenen Bäumen ist Algorithmus (4.52). Einfach sind die Fälle des Endknotens und des Knotens mit einem einzigen Nachfolger. Hat der zu löschende Knoten zwei Teilbäume, so ersetzen wir ihn wiederum durch den grössten (am weitesten rechts liegenden) Knoten seines linken Teilbaumes. Wie im Fall des Einfügens in (4.63) wird ein Boolescher Variablen-Parameter h eingeführt mit der Bedeutung "die Höhe des

Teilbaumes wurde reduziert". Ausgleichen kommt nur in Frage, wenn h den Wert true hat. Der Wert true wird h zugewiesen, wenn ein Knoten gefunden oder gelöscht wird oder wenn sich beim Ausgleichen selbst die Höhe eines Teilbaumes reduziert. In (4.64) führen wir die beiden (symmetrischen) Ausgleichsoperationen in Form von Prozeduren ein, da sie im Algorithmus zum Löschen an mehr als einer Stelle aufgerufen werden. Man beachte, dass balance1 aufgerufen wird, wenn die Höhe des linken, balance2, wenn die Höhe des rechten Astes reduziert wurde.

Die Wirkungsweise der Prozedur wird illustriert durch Fig. 4.35. Gegeben sei der ausgeglichene Baum a, dann führt schrittweises Löschen der Knoten mit den Schlüsseln 4, 8, 6, 5, 2, 1 und 7 zu den Bäumen (b) ... (h).

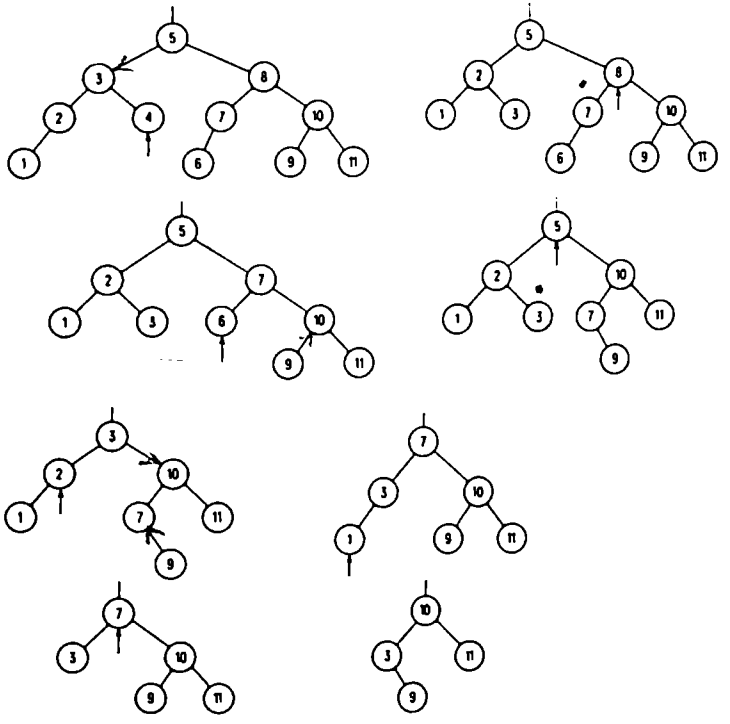


Fig. 4.35: Löschen von Elementen im ausgewogenen Baum

Löschen des Schlüssels 4 selbst ist einfach, da es ein Endknoten ist. Es führt aber zu einem unausgeglichenen Knoten 3. Die Ausgleichsoperation bedingt eine einfache LL Rotation. Ausgleichen wird wieder notwendig nach dem Löschen von Knoten 6. Dieses Mal ist der rechte Teilbaum der Wurzel (7) durch eine einfache RR Rotation auszugleichen. Löschen von Knoten 2 ist zwar selbst eine direkte Operation, da nur ein Nachfolger existiert, es bedingt aber eine komplizierte RL Doppelrotation. Der vierte Fall, eine LR Doppelrotation, wird schliesslich durch Löschen von Knoten 7 verursacht, der zunächst durch das rechteste Element seines linken Teilbaumes, d.h. durch den Knoten mit Schlüssel 3 ersetzt wurde.

```
procedure delete(x: integer; var p: ref; var h: boolean);
   var q: ref;          {h = false}                          (4.64)

   procedure balance1(var p: ref; var h: boolean);
      var p1,p2: ref; b1,b2: -1..+1;
   begin {h = true, linker Ast wurde kürzer}
      case p↑.bal of
    -1: p↑.bal := 0;
     0: begin p↑.bal := +1; h := false
        end ;
     1: begin {erneutes Ausgleichen}
           p1 := p↑.right; b1 := p1↑.bal;
           if b1 >= 0 then
           begin {einfache RR Rotation}
              p↑.right := p1↑.left; p1↑.left := p;
              if b1 = 0 then
              begin p↑.bal := +1; p1↑.bal := -1; h := false
              end else
              begin p↑.bal := 0; p1↑.bal := 0
              end ;
              p := p1
           end else
           begin {doppelte RL Rotation}
              p2 := p1↑.left; b2 := p2↑.bal;
              p1↑.left := p2↑.right; p2↑.right := p1;
              p↑.right := p2↑.left; p2↑.left := p;
              if b2 = +1 then p↑.bal := -1 else p↑.bal := 0;
              if b2 = -1 then p1↑.bal := +1 else p1↑.bal:= 0;
```

```
                p := p2; p2↑.bal := 0
            end
        end
    end
end {balance1} ;

procedure balance2(var p: ref; var h: boolean);
    var p1,p2: ref; b1,b2: -1..+1;
begin {h = true, rechter Ast wurde kürzer}
    case p↑.bal of
  1:  p↑.bal := 0;
  0:  begin p↑.bal := -1; h := false
      end ;
 -1:  begin {erneutes Ausgleichen}
          p1 := p↑.left; b1 := p1↑.bal;
          if b1 <= 0 then
          begin {einfache LL Rotation}
              p↑.left := p1↑.right; p1↑.right := p;
              if b1 = 0 then
              begin p↑.bal := -1; p1↑.bal := +1; h := false
              end else
              begin p↑.bal := 0; p1↑.bal := 0
              end ;
              p := p1
          end else
          begin {doppelte LR Rotation}
              p2 := p1↑.right; b2 := p2↑.bal;
              p1↑.right := p2↑.left; p2↑.left := p1;
              p↑.left := p2↑.right; p2↑.right := p;
              if b2 = -1 then p↑.bal := +1 else p↑.bal := 0;
              if b2 = +1 then p1↑.bal := -1 else p1↑.bal:= 0;
              p := p2; p2↑.bal := 0
          end
      end
    end
end {balance2} ;

procedure del(var r: ref; var h: boolean);
begin {h = false}
    if r↑.right <> nil then
```

```
        begin del(r↑.right,h); if h then balance2(r,h)
        end else
        begin q↑.key := r↑.key; q↑.count := r↑.count;
            r := r↑.left; h := true
        end
    end ;

begin {delete}
    if p = nil then
        begin writeln(' Schlüssel nicht im Baum'); h := false
        end else
    if x < p↑.key then
        begin delete(x,p↑.left,h); if h then balance1(p,h)
        end else
    if x > p↑.key then
        begin delete(x,p↑.right,h); if h then balance2(p,h)
        end else
    begin {entferne p↑}  q := p;
        if q↑.right = nil then
            begin p := q↑.left; h := true
            end else
        if q↑.left = nil then
            begin p := q↑.right; h := true
            end else
        begin del(q↑.left,h);
            if h then balance1(p,h)
        end ;
        {dispose(q)}
    end
end {delete}
```

Offensichtlich kann auch Löschen eines Elementes in einem ausgeglichenen Baum mit - im schlimmsten Fall - O(log n) Operationen durchgeführt werden. Ein wesentlicher Unterschied im Verhalten der Prozeduren zum Einfügen und Löschen darf aber nicht übersehen werden. Während Einfügen eines einzigen Schlüssels höchstens eine Rotation (von zwei oder drei Knoten) erfordert, kann Löschen eine Rotation für jeden Knoten entlang dem Suchpfad verursachen. Man betrachte z.B. das Löschen des rechtesten Knotens eines Fibonacci-Baumes. In diesem Fall führt das Löschen eines einzigen Knotens zu

einer Reduktion der Höhe des Baumes; Löschen des rechtesten Knotens erfordert zusätzlich die grösstmögliche Zahl von Rotationen. Dies ist deshalb die schlechteste Wahl eines Knotens im schlimmsten Fall eines ausgeglichenen Baumes, ein eher unglückliches Zusammentreffen von Zufällen! Wie wahrscheinlich sind nun Rotationen im allgemeinen? Das überraschende Resultat empirischer Tests ist, dass zwar eine Rotation bei etwa jedem zweiten Einfügen, aber nur eine Rotation bei jedem fünften Löschen erforderlich ist. Löschen in ausgeglichenen Bäumen ist daher fast ebenso leicht - oder ebenso kompliziert - wie Einfügen.

4.4.9. Optimale Suchbäume

Unsere Ueberlegungen für die Organisation von Suchbäumen gingen bisher von der Annahme aus, dass die Häufigkeit des Zugriffs für alle Knoten gleich ist, d.h. dass alle Schlüssel mit gleicher Wahrscheinlichkeit als Suchargument auftreten. Dies ist vermutlich die beste Annahme, wenn man über die Verteilung des Zugriffs nichts weiss. Es gibt aber Fälle (die zwar eher die Ausnahme als die Regel sind), in denen Information über die Wahrscheinlichkeit des Zugriffs zu verschiedenen Schlüsseln zur Verfügung steht. In diesen Fällen bleiben die Schlüssel gewöhnlich gleich, d.h. in den Suchbaum wird weder etwas eingefügt noch etwas daraus gelöscht, er hat also eine konstante Struktur. Ein typisches Beispiel ist der Scanner eines Compilers, der für jedes Wort (Bezeichner) feststellt, ob es ein Schlüsselwort (reserviertes Wort) ist. Statistische Messungen über hunderte compilierte Programme können in diesem Fall genaue Information über die relative Häufigkeit des Auftretens und damit des Zugriffs zu einzelnen Schlüsseln geben.

In einem Suchbaum sei p[i] die Wahrscheinlichkeit des Zugriffs zum Knoten i, d.h.

$$W\{x = k[i]\} = p[i], \qquad (4.65)$$
$$p[1] + p[2] + \ldots + p[n] = 1$$

Wir wollen nun den Suchbaum so organisieren, dass die gesamte Zahl der Suchschritte - über hinreichend viele Versuche gezählt - minimal wird. Dazu ändern wir die Definition der Weglänge in (4.34),

indem wir jedem Knoten ein gewisses Gewicht zuordnen. Die Knoten, zu denen häufig zugegriffen wird, werden schwere Knoten, die selten besuchten werden leichte Knoten. Die (innere) gewichtete Weglänge PI ist dann die Summe aller Wege von der Wurzel zu jedem Knoten, gewichtet mit der Wahrscheinlichkeit des Zugriffs zu diesem Knoten.

$$PI = p[1]*h[1] + p[2]*h[2] + \ldots + p[n]*h[n] \qquad (4.66)$$

h[i] ist die Stufe des Knotens i (oder seine Distanz von der Wurzel + 1). Es gilt nun, die gewichtete Weglänge für eine gegebene Verteilung der Wahrscheinlichkeit zu minimalisieren.

Man betrachte z.B. die Menge der Schlüssel 1, 2, 3 mit den Zugriffswahrscheinlichkeiten p[1] = 1/7, p[2] = 2/7 und p[3] = 4/7. Diese drei Schlüssel können auf fünf verschiedene Arten als Suchbäume dargestellt werden (vgl. Fig. 4.36).

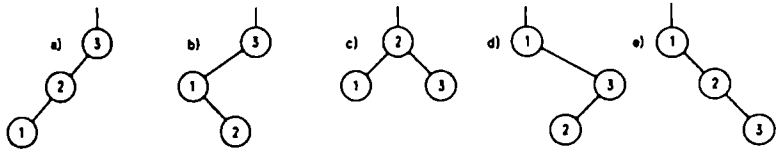


Fig. 4.36: Geordnete Bäume mit drei Elementen

Die gewichteten Weglängen berechnen sich nach (4.66) zu

(a) PI = (1*3 + 2*2 + 4*1)/7 = 11/7
(b) PI = (1*2 + 2*3 + 4*1)/7 = 12/7
(c) PI = (1*2 + 2*1 + 4*2)/7 = 12/7
(d) PI = (1*1 + 2*3 + 4*2)/7 = 15/7
(e) PI = (1*1 + 2*2 + 4*3)/7 = 17/7

Damit ist in diesem Beispiel nicht der völlig ausgeglichene, sondern der degenerierte Baum (a) die beste Anordnung.

Das Beispiel des Compiler-Scanner bringt einem sofort auf die Idee, dieses Problem unter einer etwas allgemeineren Bedingung zu sehen: Im Quellentext vorkommende Worte sind nicht immer Schlüsselworte; tatsächlich sind Schlüsselworte eher die Ausnahme. Das

Herausfinden, dass ein gegebenes Wort kein Schlüssel im Suchbaum ist, kann man als Zugriff zu einem hypothetischen "speziellen Knoten" betrachten, der zwischen dem nächst kleineren und nächst höheren Schlüssel (vgl. Fig. 4.19) mit einer zugeordneten äusseren Weglänge eingefügt ist. Wenn ausserdem die Wahrscheinlichkeit q[i] bekannt ist, dass ein gesuchtes Argument x zwischen den beiden Schlüsseln k[i] und k[i+1] liegt, kann diese Information die Struktur des optimalen Suchbaumes wesentlich verändern. Somit verallgemeinern wir das Problem durch die Berücksichtigung nicht erfolgreichen Suchens (Fig. 4.37).

Die gesamte gewichtete mittlere Weglänge ist nun

$$P = \sum_{i=1}^{n} p[i]*h[i] + \sum_{j=1}^{m} q[j]*h'[j] \qquad (4.67)$$

wobei

$$\sum_{i=1}^{n} p[i] + \sum_{j=1}^{m} q[j] = 1$$

h[i] ist die Stufe des (inneren) Knotens i und h'[j] die Stufe des äusseren Knotens j. Die mittlere gewichtete Weglänge kann als "Preis" für den Suchbaum aufgefasst werden, da sie ein Mass für den beim Suchen anfallenden Aufwand darstellt. Der Suchbaum, dessen Struktur die kleinsten Kosten von allen Bäumen für eine gegebene Menge von Schlüsseln und Wahrscheinlichkeiten p[i] und q[j] verursacht, heisst optimaler Baum.

Um den optimalen Baum zu finden, muss die Summe der p's und q's nicht gleich 1 sein. Tatsächlich werden diese Wahrscheinlichkeiten normalerweise in Experimenten bestimmt, indem man die Zugriffe zu Knoten zählt. Statt der Wahrscheinlichkeiten p[i] und q[j] werden wir folglich solche Zähler der Häufigkeit verwenden und sie bezeichnen mit

a[i] = Anzahl der Fälle mit Suchargument x gleich k[i]
b[j] = Anzahl der Fälle, in denen das Suchargument x zwischen k[j] und k[j+1] liegt.

Nach Konvention gibt b[0] an, wie oft x kleiner ist als k[1], und b[n], wie oft x grösser ist als k[n] (vgl. Fig. 4.37). Im folgenden werden wir P zur Bezeichnung der kumulierten gewichteten Weglänge statt der mittleren Weglänge verwenden.

$$P = \sum_{i=1}^{n} a[i]*h[i] + \sum_{j=1}^{m} b[j]*h'[j] \qquad (4.68)$$

Wir vermeiden dadurch die Berechnung der Wahrscheinlichkeiten aus gemessenen Häufigkeiten und haben zusätzlich den Vorteil, dass wir uns bei der Suche nach dem optimalen Baum auf ganze Zahlen beschränken können.

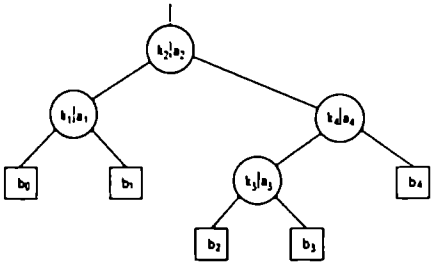


Fig. 4.37: Geordneter Baum mit Zugriffshäufigkeiten

Berücksichtigt man die Tatsache, dass die Zahl der möglichen Anordnungen von n Knoten exponentiell mit n wächst, so scheint die Aufgabe, das Optimum zu finden, für grosses n ziemlich hoffnungslos zu sein. Optimale Bäume haben aber eine bemerkenswerte Eigenschaft, die bei der Suche nützlich ist. Alle Teilbäume sind auch optimal. Wenn z.B. der Baum in Fig. 4.37 für gegebene a's und b's optimal ist, so ist der Teilbaum mit den Schlüsseln k[3] und k[4] auch optimal. Diese Eigenschaft empfiehlt einen Algorithmus, der systematisch immer grössere Bäume findet, ausgehend von einzelnen Knoten als kleinstmögliche Teilbäume. Der Baum wächst also "von den Blättern zur Wurzel", was bei unserer verkehrten Art, die Bäume zu zeichnen, die Richtung von "unten nach oben" ist [16].

Die Gleichung (4.69) ist der Schlüssel zu diesem Algorithmus. Sei P

die gewichtete Weglänge eines Baumes, und seien P[L] und P[R] die des linken und rechten Teilbaumes seiner Wurzel, so ist P die Summe von P[L] und P[R] und der Anzahl von Suchdurchläufen auf dem einen Ast zu der Wurzel, was einfach die gesamte Zahl W solcher Suchläufe ist.

$$P = P[L] + W + P[R] \tag{4.69}$$

$$W = \sum_{i=1}^{n} a[i] + \sum_{j=1}^{m} b[j] \tag{4.70}$$

Wir nennen W das Gewicht des Baumes. Seine mittlere Weglänge ist dann P/W.

Diese Ueberlegungen zeigen die Notwendigkeit einer Notation für die Gewichte und Weglängen irgendeines Teilbaumes, der aus einer Zahl benachbarter Schlüssel besteht. Wir wollen mit w[i,j] das Gewicht und mit p[i,j] die Weglänge des optimalen Teilbaumes T[i,j] bezeichnen, der aus den Knoten mit den Schlüsseln k[i+1], k[i+2], ... , k[j] besteht. Diese Grössen werden durch die Rekursions-Relationen (4.71) und (4.72) definiert.

$$\begin{aligned} w[i,i] &= b[i] && (0 <= i <= n) \\ w[i,j] &= w[i,j-1] + a[j] + b[j] && (0 <= i < j <= n) \end{aligned} \tag{4.71}$$

$$\begin{aligned} p[i,i] &= w[i,i] && (0 <= i <= n) \\ p[i,j] &= w[i,j] + \min_{i<k<=j} (p[i,k-1] + p[k,j]) && (0 <= i < j <= n) \end{aligned} \tag{4.72}$$

Die letzte Gleichung ergibt sich unmittelbar aus (4.69) und der Definition der Optimalität.

Da es ungefähr n**2/2 Werte p[i,j] gibt und da (4.72) eine Wahl zwischen 0 < j-1 <= n Fällen zulässt, umfasst die Operation des Minimalisierens ungefähr (n**3)/6 Operationen. Knuth weist darauf hin, dass durch folgende Ueberlegungen ein Faktor n eingespart werden kann, was allein diesen Algorithmus für praktische Zwecke rettet.

Sei r[i,j] ein Wert für k, der in (4.72) ein Minimum ergibt. Man kann die Grenzen für die Suche von r[i,j] auf ein sehr viel kleineres Intervall beschränken, d.h. man kann die Zahl der j-i Auswertungsschritte reduzieren. Der Schlüssel liegt in der Beobachtung, dass, wenn wir die Wurzel r[i,j] des optimalen Teilbaumes T[i,j] gefunden haben, weder ein Ausdehnen des Baumes durch Hinzufügen eines Knotens auf der rechten Seite noch das Entfernen des äussersten linken Knotens jemals ein nach links Wandern der Wurzel bewirken kann. Dies wird ausgedrückt durch folgende Relation

```
r[i,j-1] <= r[i,j] <= r[i+1,j]                                    (4.73)
```

die die Suche nach möglichen Lösungen für r[i,j] auf den Bereich r[i,j-1] ... r[i+1,j] beschränkt und insgesamt O(n**2) elementare Schritte benötigt. Wir sind nun in der Lage, den Algorithmus zur Optimierung im Detail zu erarbeiten. Erinnern wir uns an folgende Definitionen, die sich auf optimale Bäume T[i,j] aus Knoten mit Schlüsseln k[i+1] ... k[j] beziehen.

1. a[i] Häufigkeit der Suche nach k[i]
2. b[j] Häufigkeit der Suche eines Argumentes x zwischen k[j] und k[j+1]
3. w[i,j] Gewicht von T[i,j]
4. p[i,j] gewichtete Weglänge von T[i,j]
5. r[i,j] Index der Wurzel von T[i,j] .

Mit gegebenen

```
type index = 0..n
```

vereinbaren wir folgende Arrays:

```
a : array [1..n] of integer;                                      (4.74)
b : array [index] of integer;
p,w: array [index,index] of integer;
r : array [index,index] of index
```

Man nehme an, dass das Gewicht w[i,j] auf direkte Art aus a und b berechnet wurde (vgl. (4.71)). Nun betrachte man w als Argument der zu entwickelnden Prozedur und r als ihr Resultat, das ihre Struktur vollständig beschreibt. p kann als Zwischenresultat angesehen werden. Beginnend mit der Betrachtung der kleinstmöglichen Teil-

bäume, nämlich denen ohne jeden Knoten, gehen wir weiter zu immer grösser werdenden Bäumen. Wir bezeichnen die Breite j-i des Teilbaumes T[i,j] mit h. Dann können wir die Werte p[i,i] für alle Bäume mit h = 0 entsprechend (4.72) sehr einfach bestimmen.

```
for i := 0 to n do p[i,i] := w[i,i]                    (4.75)
```

Wenn h = 1, haben wir es mit Bäumen mit nur einem Knoten zu tun, der gleichzeitig auch die Wurzel ist (Fig. 4.38).

```
for i := 0 to n-1 do                                   (4.76)
   begin j := i+1; p[i,j] := p[i,i]+p[j,j]; r[i,j] := j
   end
```

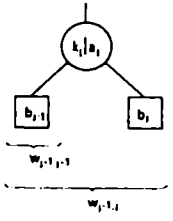


Fig. 4.38: Optimaler Baum mit einem Element

Man beachte, dass i die linke und j die rechte Grenze des Index in dem betrachteten Baum T[i,j] bezeichnet. Für die Fälle h > 1 verwenden wir eine Schleifenanweisung mit Werten von h von 2 bis n, der Fall h = n umfasst den ganzen Baum T[0,n]. Die minimale Weglänge p[i,j] und der Index r[i,j] der entsprechenden Wurzel werden jedesmal durch eine einfache wiederholte Anweisung mit einem über das durch (4.73) gegebene Intervall laufenden Index berechnet.

```
for h := 2 to n do                                     (4.77)
for i := 0 to n-h do
begin j := i+h;
   "finde m und min = minimum(p[i,m-1]+p[m,j])
    für alle m mit r[i,j-1] <= m <= r[i+1,j]";
   p[i,j] := min + w[i,j];  r[i,j] := m
end
```

Die Einzelheiten der Verfeinerung der in Anführungszeichen stehenden Anweisung können Programm 4.6 entnommen werden. Die mittlere Weglänge von T[0,n] ist nun gegeben durch den Quotienten

p[0,n] / w[0,n]

Seine Wurzel ist der Knoten mit dem Index r[0,n] und seine beiden Teilbäume sind T[0,r[0,n]] und T[r[0,n],n] , usw.

Aus dem Algorithmus (4.77) geht hervor, dass der Aufwand zur Bestimmung der optimalen Struktur O(n**2) ist; die Grösse des benötigten Speicherplatzes ist ebenfalls O(n**2). Dies ist für sehr grosses n nicht annehmbar, und Algorithmen mit grösserer Effizienz sind besonders wünschenswert. Einer davon ist der von Hu und Tucker [14] entwickelte Algorithmus, der nur O(n) Speicherplatz und O(n*log(n)) Operationen benötigt. Er behandelt aber nur die Fälle, in denen die Häufigkeit der Schlüssel null ist (a[i] = 0), d.h. bei denen nur vergebliche Versuche registriert werden. Ein anderer Algorithmus, der ebenfalls O(n) Speichereinheiten und O(n*log(n)) Berechnungen benötigt, wurde von Walker und Gotlieb [29] beschrieben. Statt das Optimum zu finden, verspricht dieser Algorithmus nur einen nahezu optimalen Baum. Er kann sich daher auf heuristische Prinzipien stützen. Die zugrunde liegende Idee ist folgende.

Man betrachte die Knoten (echte und spezielle) als auf einer linearen Skala verteilt, gewichtet durch ihre Häufigkeit (oder Wahrscheinlichkeit) des Zugriffs. Dann bestimme man den Knoten, der dem "Schwerpunkt" am nächsten liegt. Dieser Knoten heisst Zentroid und hat den Index

$$(1/w) * \left(\sum_{i=1}^{n} i*a[i] + \sum_{j=1}^{n} j*b[j] \right) \qquad (4.78)$$

gerundet auf die nächste ganze Zahl. Haben alle Knoten gleiches Gewicht, so fällt die Wurzel des gesuchten optimalen Baumes natürlich mit dem Zentroid zusammen und - so wird argumentiert - wird in den meisten Fällen in enger Nachbarschaft des Zentroid sein. Eine begrenzte Suche wird dann zur Bestimmung des lokalen Optimums verwendet, und danach wird diese Prozedur auf die so entstehenden

beiden Teilbäume angewendet. Die Wahrscheinlichkeit, dass die Wurzel sehr nahe beim Zentroid liegt, wächst mit der Grösse n des Baumes. Sobald die Teilbäume eine "handliche" Grösse erreicht haben, kann ihr Optimum durch obigen exakten Algorithmus bestimmt werden.

4.4.10. Ausgabe einer Baumstruktur

Wir wenden uns nun dem verwandten Programmierproblem zu, wie eine Ausgabe zu generieren ist, die die Struktur des Baumes in relativ klarer graphischer Form nur mit Hilfe eines normalen Druckers aufzeichnet. Das heisst, wir wollen ein Bild des Baumes zeichnen, indem wir die Schlüssel als Knoten drucken und sie durch geeignete horizontale und vertikale Striche verbinden (wir werden dazu das Minuszeichen und den Doppelpunkt aus dem reduzierten ASCII Zeichensatz verwenden).

Bei einem Zeilendrucker, dessen Daten wir als Text-File darstellen, d.h. als Sequenz von Zeichen, können wir nur streng sequentiell von links nach rechts und von oben nach unten vorgehen Daher scheint es vernünftig, zuerst eine Darstellung des Baumes mit möglichst genauer Wiedergabe seiner topologischen Struktur aufzubauen. Der zweite Schritt ist dann die Abbildung dieser Struktur auf ein Druckerblatt und die Berechnung der genauen Koordinaten der Knoten und Verbindungen.

Für die erste Aufgabe können wir von unserer Erfahrung mit Algorithmen zur Erzeugung von Bäumen zehren und werden zweifellos zur Lösung des rekursiv definierten Problems eine rekursive Prozedur wählen. Wir formulieren eine Funktionsprozedur tree, ähnlich zu der in Programm 4.3 vorkommenden. Die Parameter i und j sind die begrenzenden Indizes für die zum Baum gehörenden Knoten. Seine Wurzel ist dann definiert als der Knoten mit Index r[i,j]. Bevor wir aber weiterfahren, müssen wir den Typ der die Knoten darstellenden Variablen definieren. Er muss zwei Zeiger zu den Teilbäumen und den Schlüssel des Knotens enthalten. Aus Gründen, die im zweiten Schritt erörtert werden, werden zwei zusätzliche Felder namens pos und link eingeführt. Die gewählten Definitionen werden in (4.79) gezeigt, und die sich ergebende Funktionsprozedur ist als Programm

4.6 gelistet.

```
type ref = ↑node;                                          (4.79)
     node = record key: alfa;
                   pos: lineposition;
                   left,right,link: ref
            end
```

Man beachte, dass diese Prozedur die Zahl der generierten Knoten in einer globalen Variablen k zählt. Dem k-ten Knoten wird der k-te Schlüssel zugewiesen, und da die Schlüssel alphabetisch geordnet sind, ergibt die Multiplikation von k mit einem konstanten Skalierungsfaktor die horizontale Koordinate jedes Schlüssels, ein Wert, der sofort mit der anderen Information gespeichert wird. Man beachte auch, dass wir von der Konvention abgewichen sind, ganze Zahlen als Schlüssel zu verwenden, und sie als vom Typ alfa annahmen, der einen Array von Zeichen mit fester (maximaler) Länge darstellt, auf dem eine alphabetische Ordnung definiert ist.

Fig. 4.39 gibt ein Bild des bisher Erreichten. Sind die Menge der n Schlüssel und die berechnete Matrix r[i,j] gegeben, so generieren die Anweisungen

```
k := 0; root := tree(0,n)
```

die vorläufig verknüpfte Baumstruktur mit Angabe der horizontalen Position der Knoten und ihrer durch ihre Stufe im Baum implizit bestimmten vertikalen Position.

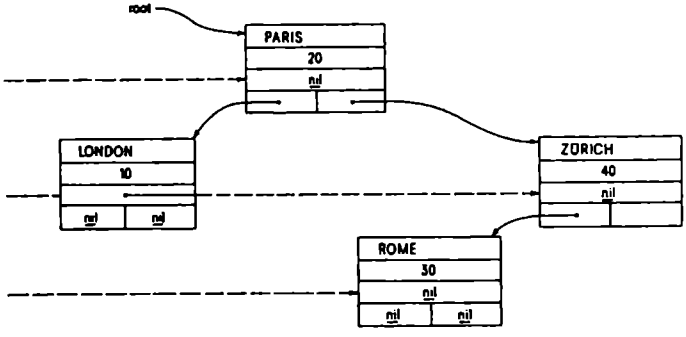


Fig. 4.39: Teil eines durch Programm 4.6 erzeugten Baumes

Wir kommen nun zu Schritt 2: der Abbildung des Baumes auf Papier. Dabei müssen wir strikt von der Wurzel aus nach unten vorgehen und bei jedem Schritt eine Zeile von Knoten verarbeiten. Wie greifen wir aber zu den auf einer Zeile liegenden Knoten zu? Für diesen Zweck, nämlich zum Verketten der Knoten in der gleichen Zeile, haben wir bereits das Record-Feld link eingeführt. Die zu erstellenden Ketten sind in Fig. 4.39 als gestrichelte Verbindungen eingetragen. Bei jedem Schritt der Verarbeitung setzen wir die Existenz einer Kette voraus, die die zu druckenden Knoten verbindet - wir nennen diese momentan benutzte Kette current - und während der Behandlung jedes Knotens identifizieren wir seine Nachfolger (falls vorhanden) und verknüpfen sie in einer zweiten Kette - der Kette next. Wenn wir eine Stufe nach unten gehen, wird die Kette next zur Kette current, und die Kette next wird als leer markiert.

Die Einzelheiten des Algorithmus können Programm 4.6 entnommen werden. Die folgenden Bemerkungen sollen der Verdeutlichung einiger Punkte dienen:

1. Die Ketten von Knoten auf einer Zeile werden von links nach rechts generiert, wobei der äusserste linke Knoten der letzte wird. Da die Knoten in der gleichen Reihenfolge zu bearbeiten sind, muss die Liste umgekehrt werden. Diese Umkehrung erfolgt, wenn die Liste next zu current wird.

2. Eine Druckzeile mit den Schlüsseln - genannt Hauptzeile - enthält auch die horizontalen Verbindungen (siehe Fig. 4.40). Die Variablen u1, u2, u3 und u4 bezeichnen die Anfangs- und Endpositionen der linken und rechten horizontalen Verbindungen eines Knotens.

3. Der Konstruktion jeder Hauptzeile gehen drei Zeilen mit den vertikalen Teilen der Verbindungen voran.

Wir wollen nun die Struktur von Programm 4.6 beschreiben. Seine beiden Hauptteile sind die Prozeduren zum Erstellen eines optimalen Suchbaumes für eine gegebene Verteilung w der Gewichte und zur Darstellung eines Baumes für gegebene Indizes r. Das ganze Programm ist ausgerichtet auf die Behandlung von Programmtexten, speziell PASCAL-Programmen. Im ersten Teil wird solch ein Programm gelesen,

seine Bezeichner und Schlüsselworte werden erfasst und ergeben die Zähler a[i] und b[i] zum Finden eines Schlüsselwortes k[i] oder eines Bezeichners zwischen k[j] und k[j+1]. Nach dem Drucken einer Statistik der Häufigkeiten berechnet das Programm die Weglänge des völlig ausgeglichenen Baumes und bestimmt nebenher die Wurzeln seiner Teilbäume. Danach wird die mittlere gewichtete Weglänge gedruckt und der Baum abgebildet.

Im dritten Teil wird die Prozedur optree zur Berechnung des optimalen Suchbaumes aktiviert; dieser wird anschliessend auch ausgegeben. Schliesslich werden die gleichen Prozeduren zur Berechnung und Abbildung des Baumes verwendet, der optimal ist, wenn allein die Häufigkeiten der Schlüssel berücksichtigt werden.

Tabelle 4.5 und Fig. 4.40 bis 4.42 zeigen die von Programm 4.6 erzeugten Resultate bei der Anwendung auf den Programmtext selbst. Die Unterschiede der drei Abbildungen zeigen, dass der ausgeglichene Baum auch nicht als nur nahezu optimal betrachtet werden kann und dass die Häufigkeiten der Nicht-Schlüsselwörter die Wahl der optimalen Struktur entscheidend beeinflussen.

```
Schlüssel und Häufigkeiten:
       4          7 ARRAY
      14         27 BEGIN
      19          0 CASE
      15          2 CONST
       8          5 DIV
       0          0 DOWNTO
       0         20 DO
       0          8 ELSE
       0         28 END
       1          0 FILE
       0         12 FOR
       0          2 FUNKTION
       0          0 GOTO
       9         13 IF
      23          2 IN
     200          0 LABEL
      22          0 MOD
      17         10 NIL
```

24	7	OF
17	2	PROCEDURE
0	1	PROGRAM
53	1	RECORD
6	8	REPEAT
16	0	SET
10	13	THEN
0	12	TO
6	2	TYPE
1	8	UNTIL
39	5	VAR
0	8	WHILE
0	0	WHITH
37		
549	203	

Tabelle 4.5

```
program optimaltree(input,output);
   const n = 31; {Anzahl Schlüssel}
      kln = 10;  {max. Schlüssellänge}
   type index = 0..n;
      alfa = packed array [1..kln] of char;
   var ch: char;
       k1, k2: integer;
       id: alfa;     {Bezeichner oder Schlüssel}
       buf: array [1..kln] of char;    {Zeichenpuffer}
       key: array [1..n] of alfa;
       i,j,k: integer;
       a: array [1..n] of integer;
       b: array [index] of integer;
       p,w: array [index,index] of integer;
       r: array [index,index] of index;
       suma, sumb: integer;

   function baltree(i,j: index): integer;
      var k: integer;
   begin k := (i+j+1) div 2; r[i,j] := k;
```

```
   if i >= j then baltree := b[k] else
      baltree := baltree(i,k-1) + baltree(k,j) + w[i,j]
end {baltree} ;

{ ---------------------------------------------------------- }

procedure opttree;
   var x, min: integer;
      i,j,k,h,m: index;
begin {Argument: w, Resultat: p,r}
   for i := 0 to n do p[i,i] := w[i,i];
      {Breite des Baumes h = 0}
   for i := 0 to n-1 do
      {Breite des Baumes h = 1}
   begin j := i+1;
      p[i,j] := p[i,i] + p[j,j];  r[i,j] := j
   end ;
   for h := 2 to n do     { h = Breite des betrachteten Baumes}
   for i := 0 to n-h do   { i = linker Index dieses Baumes}
   begin  j := i+h;       { j = rechter Index dieses Baumes}
      m := r[i,j-1];  min := p[i,m-1] + p[m,j];
      for k := m+1 to r[i+1,j] do
      begin x := p[i,k-1] + p[k,j];
         if x < min then
            begin m := k;  min := x
            end
      end ;
      p[i,j] := min + w[i,j];  r[i,j] := m
   end ;
end {opttree} ;

{ ---------------------------------------------------------- }

procedure printtree;
   const lw = 120;   {Zeilenbreite des Druckers}
   type ref = ↑node;
      lineposition = 0..lw;
      node = record key: alfa;
                    pos: lineposition;
                    left, right, link: ref
```

```
            end ;
   var root, current, next: ref;
      q,q1,q2: ref;
      i: integer;
      k: integer;
      u, u1, u2, u3, u4: lineposition;

   function tree(i,j: index): ref;
      var p: ref;
   begin if i = j then p := nil else
      begin new(p);
         p↑.left := tree( i, r[ i,j]-1);
         p↑.pos := trunc(( lw-kln)*k/(n-1)) + (kln div 2);
         k := k+1;
         p↑.key := key[r[ i,j]];
         p↑.right := tree(r[ i,j], j)
      end ;
      tree := p
   end ;

begin k := 0;  root := tree(0,n);
   current := root; root↑.link := nil;
   next := nil;
   while current <> nil do
   begin {arbeite nach unten;
          drucke zuerst vertikale Linien}
      for i := 1 to 3 do
      begin u := 0; q := current;
         repeat u1 := q↑.pos;
            repeat write(' '); u := u+1
            until u = u1;
            write(':'); u := u+1; q := q↑.link
         until q = nil;
         writeln
      end ;
      {drucke Hauptzeile; ausgehend von den Knoten der
       momentanen Liste werden deren Nachfolger gesammelt
       und bilden die nächste Liste}
      q := current; u := 0;
      repeat unpack(q↑.key, buf, 1);
```

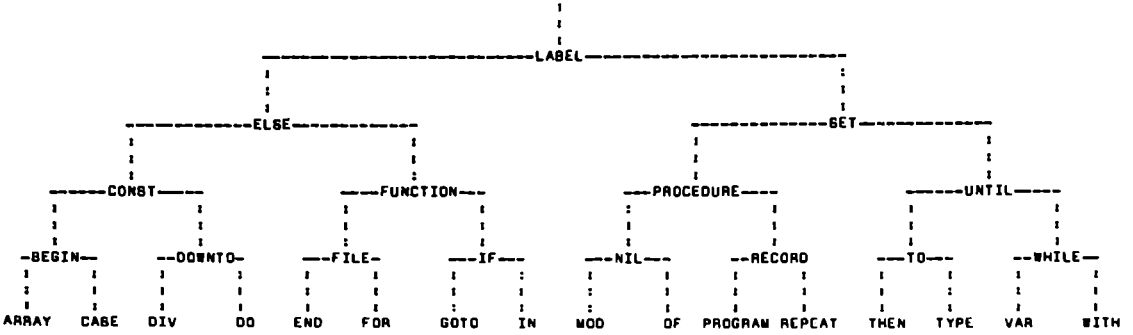
```
        {zentriere Schlüssel bzgl. pos}  i := kln;
        while buf[i] = ' ' do  i := i-1;
        u2 := q↑.pos - ((i-1) div 2); u3 := u2+i;
        q1 := q↑.left; q2 := q↑.right;
        if q1 = nil then u1 := u2 else
           begin u1 := q1↑.pos; q1↑.link := next; next := q1
           end ;
        if q2 = nil then u4 := u3 else
           begin u4 := q2↑.pos+1; q2↑.link := next;
              next := q2
           end ;
        i := 0;
        while u < u1 do begin write(' '); u := u+1 end ;
        while u < u2 do begin write('-'); u := u+1 end ;
        while u < u3 do
           begin i := i+1; write(buf[i]); u := u+1 end ;
        while u < u4 do begin write('-'); u := u+1 end ;
        q := q↑.link
     until q = nil;
     writeln;
     {invertiere nächste Liste und mache sie zur momentanen}
     current := nil;
     while next <> nil do
        begin q := next; next := q↑.link;
           q↑.link := current; current := q
        end
  end
end {printtree} ;

{ ------------------------------------------------------------ }

begin {initialisiere Tabellen der Schlüssel und Zähler}
   key[ 1] := 'array     '; key[ 2] := 'begin     ';
   key[ 3] := 'case      '; key[ 4] := 'const     ';
   key[ 5] := 'div       '; key[ 6] := 'downto    ';
   key[ 7] := 'do        '; key[ 8] := 'else      ';
   key[ 9] := 'end       '; key[10] := 'file      ';
   key[11] := 'for       '; key[12] := 'function  ';
   key[13] := 'goto      '; key[14] := 'if        ';
   key[15] := 'in        '; key[16] := 'label     ';
```

```
key[17] := 'mod       '; key[18] := 'nil       ';
key[21] := 'program   '; key[22] := 'record    ';
key[23] := 'repeat    '; key[24] := 'set       ';
key[27] := 'type      '; key[28] := 'until     ';
key[29] := 'var       '; key[30] := 'while     ';
key[31] := 'with      ';
for i := 1 to n do
   begin a[i] := 0;  b[i] := 0
   end ;
b[0] := 0; k2 := kln;
{durchsuche Eingabetext und bestimme a und b}
while not eof(input) do
begin read(ch);
   if ch in ['a'..'z'] then
   begin {Bezeichner oder Schlüssel} k1 := 0;
      repeat if k1 < kln then
             begin k1 := k1+1; buf[k1] := ch
             end ;
         read(ch)
      until not (ch in ['a'..'z', '0'..'9']);
      if k1 >= k2 then k2 := k1 else
      repeat buf[k2] := ' '; k2 := k2-1
      until k2 = k1;
      pack(buf,1,id);
      i := 1; j := n;
      repeat k := (i+j) div 2;
         if key[k] <= id then i := k+1;
         if key[k] >= id then j := k-1;
      until i > j;
      if key[k] = id then a[k] := a[k] + 1 else
         begin k := (i+j) div 2; b[k] := b[k]+1
         end ;
   end else
   if ch = '''' then
      repeat read(ch) until ch = '''' else
   if ch = '{' then
      repeat read(ch) until ch = '}'
end ;
writeln(' keys and frequencies of occurrence:');
suma := 0; sumb := b[0];
```

```
   for i := 1 to n do
   begin suma := suma+a[i]; sumb := sumb+b[i];
      writeln(b[i-1], a[i], ' ', key[i])
   end ;
   writeln(b[n]);
   writeln('    ------    ------');
   writeln(sumb, suma);
   {berechne w aus a und b}
   for i := 0 to n do
   begin w[i,i] := b[i];
      for j := i+1 to n do w[i,j] := w[i,j-1] + a[j] + b[j]
   end ;
   writeln;
   write(' average path length of balanced tree =');
   writeln(baltree(0,n)/w[0,n]:6:3); printtree;
   opttree;
   writeln;
   write(' average path length of optimal tree =');
   writeln(p[0,n]/w[0,n]:6:3);  printtree;
   {betrachte nur noch Schlüssel durch setzen von b = 0}
   for i := 0 to n do
   begin w[i,i] := 0;
      for j := i+1 to n do w[i,j] := w[i,j-1] + a[j]
   end ;
   opttree;
   writeln;
   writeln(' optimal tree considering keys only');
   printtree
end .
```

Programm 4.6

MITTLERE PFADLAENGE DES AUSGEWOGENEN BAUMES = 5.566



MITTLERE PFADLAENGE DES OPTIMALEN BAUMES = 4.168

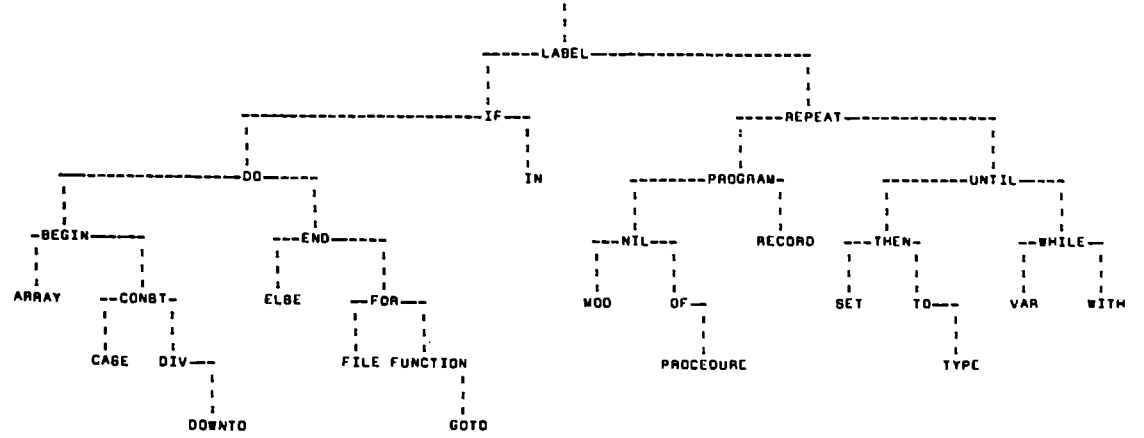
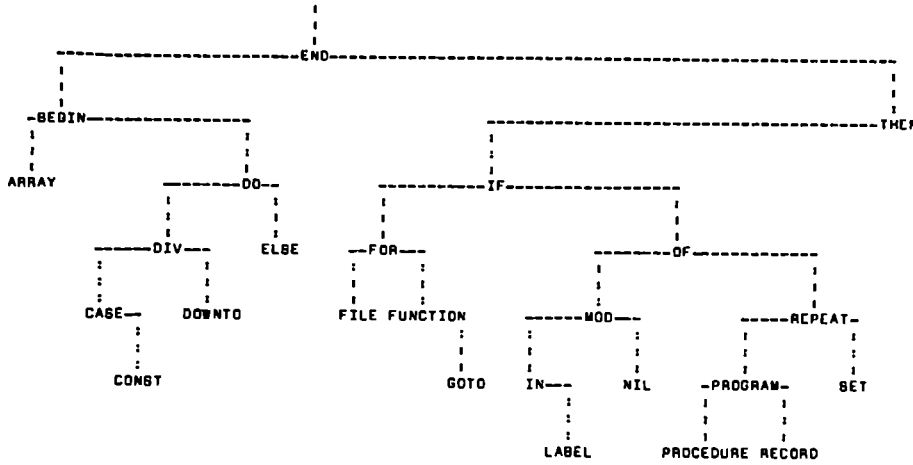


Fig. 4.40 und 4.41

Fig. 4.42

OPTIMALER BAUM BEI AUSSCHLIESSLICHER BERUECKSICHTIGUNG DER SCHLUESSELWOERTER



4.5. Vielweg-Bäume

Bisher haben wir unsere Diskussion auf Bäume beschränkt, deren Knoten höchstens zwei Nachfolger hatten, d.h. auf binäre Bäume. Dies ist völlig ausreichend, wenn wir z.B. die familiären Beziehungen vorzugsweise in der Art eines "Stammbaumes" darstellen, d.h. jede Person mit ihren Eltern in Verbindung bringen. Niemand hat nämlich mehr als zwei Elternteile! Was aber, wenn jemand die Blickrichtung auf die "Nachkommenschaft" bevorzugt? Er muss berücksichtigen, dass manche Leute mehr als zwei Kinder haben, und sein Baum wird Knoten mit mehreren Aesten enthalten. Mangels eines besseren Ausdrucks nennen wir sie Vielweg-Bäume.

Diese Strukturen sind natürlich nichts besonderes, und wir kennen bereits alle Hilfsmittel der Programmierung und Datendefinition, um diese Situation zu meistern. Ist z.B. eine absolute obere Schranke für die Zahl der Kinder gegeben (was zugegebenermassen eine futuristische Annahme ist), so kann man die Kinder als eine Array-Komponente des für eine Person stehenden Records darstellen. Schwankt jedoch die Zahl der Kinder zwischen verschiedenen Personen sehr stark, so kann dies zu einer schwachen Ausnutzung des verfügbaren Speichers führen. In diesem Fall wird man die Nachkommen besser in einer linearen Liste anordnen mit einem den Eltern zugewiesenen Zeiger auf den jüngsten (oder ältesten) Nachkommen. (4.80) zeigt eine mögliche Typ-Definition für diesen Fall und Fig. 4.43 eine entsprechende Datenstruktur.

```
type person = record name: alfa;                    (4.80)
                     Geschwister: ↑person;
                     Nachkommen:  ↑person
              end
```

Wir stellen nun fest, dass wir dieses Bild nur um 45 Grad zu drehen haben, um einen perfekten binären Baum zu bekommen. Diese Ansicht täuscht aber, denn die Bedeutung der beiden Zeiger ist völlig verschieden. Man behandelt normalerweise Geschwister nicht ungestraft wie Nachkommen und sollte es daher auch in der Definition der Daten nicht tun. Dieses Beispiel könnte ebenso leicht zu einer komplizierteren Datenstruktur durch Einführen weiterer Komponenten

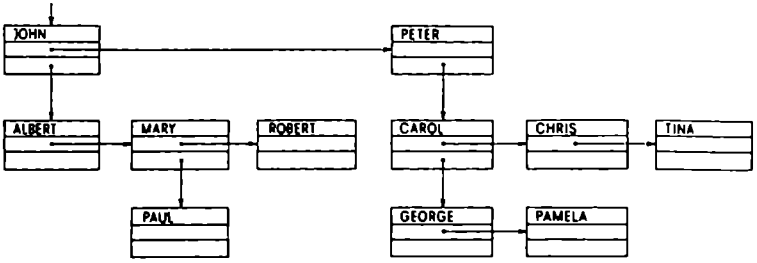


Fig. 4.43: Verkettete Darstellung eines Baumes höherer Ordnung

im Record einer Person zur Darstellung zusätzlicher familiärer Beziehungen erweitert werden. Ein möglicher Kandidat für eine im allgemeinen nicht aus der Beziehung von Geschwistern und Nachkommen ableitbare Beziehung ist die von Mann und Frau oder auch die umgekehrte Beziehung Vater und Mutter. Solch eine Struktur wächst rasch zu einer komplexen "Verwandtschafts-Datenbank" (relational data bank), und mehrere Bäume können darauf abgebildet werden. Die Algorithmen, die auf solchen Strukturen arbeiten, sind eng verknüpft mit den Datendefinitionen, und es ist nicht sinnvoll, allgemeine Regeln oder weitverbreitete und weitgehend anwendbare Techniken anzugeben.

Es gibt aber ein sehr praktisches Gebiet für die Anwendung von Vielweg-Bäumen, das von allgemeinem Interesse ist. Es handelt sich um die Erstellung und den Unterhalt von Suchbäumen grossen Ausmasses, bei denen Einfügen und Löschen notwendig ist, wo aber der Vordergrundspeicher einer Rechenanlage zur Speicherung über lange Zeit nicht gross genug oder zu teuer ist.

Die Knoten eines Baumes seien dann auf einem Hintergrundspeicher, etwa einem Plattenspeicher, zu speichern. Dynamische Datenstrukturen, wie sie in diesem Kapitel eingeführt wurden, eignen sich besonders für die Einbeziehung von Hintergrundspeicher. Die wesentliche Neuerung ist lediglich, dass Zeiger durch Platten- statt Hauptspeicheradressen dargestellt werden. Die Verwendung eines binären Baumes für eine Datenmenge von, z.B., einer Million Elementen erfordert im Mittel etwa log 10**6 = 20 Suchschritte. Da jeder

Schritt jetzt einen Plattenzugriff (mit entsprechender Latenzzeit) einschliesst, ist eine Speicherorganisation mit weniger Zugriffen äusserst wünschenswert. Der Vielweg-Baum ist eine perfekte Lösung für dieses Problem. Wird zu einem auf einem Hintergrundspeicher liegenden Element zugegriffen, so kann genauso gut zu einer ganzen Gruppe ohne wesentliche zusätzliche Kosten zugegriffen werden. Dies legt nahe, den Baum in Teilbäume zu unterteilen und Teilbäume als gleichzeitig zugreifbare Einheiten darzustellen. Wir werden diese Teilbäume Seiten (page) nennen. Fig. 4.44 zeigt einen in Seiten aufgeteilten binären Baum, wobei jede Seite aus 7 Knoten besteht.

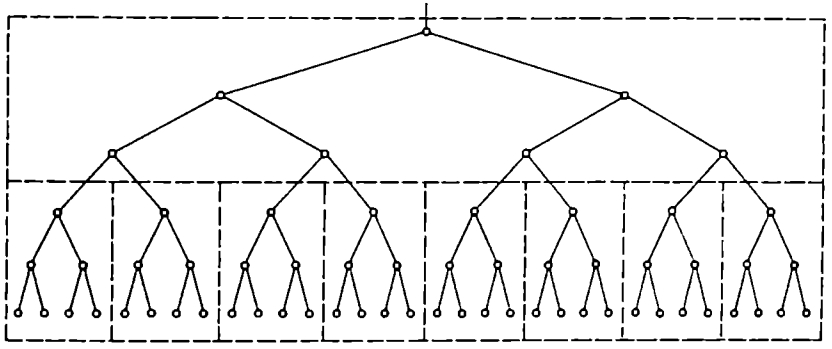


Fig. 4.44: In Abschnitte unterteilter binärer Baum

Die Ersparnis bei der Zahl der Speicherzugriffe - jeder Zugriff zu einer Seite bedingt jetzt einen Zugriff zur Platte - kann beträchtlich sein. Setzen wir z.B. 100 Knoten auf eine Seite (dies ist ein vernünftiger Wert), so wird die Suche mit einer Million Elementen im Mittel nur log100(10**6) = 3 Seitenzugriffe statt 20 erfordern. Lässt man den Baum jedoch "willkürlich" wachsen, können es im schlimmsten Fall aber immer noch 10**4 sein! Daher ist in diesem Fall ein Schema für das kontrollierte Wachsen des Vielweg-Baumes unbedingt erforderlich.

4.5.1. B-Bäume

Bei der Suche nach einem Kriterium für kontrolliertes Wachstum wird das des vollständigen Ausgleichens schnell ausscheiden, da es

zuviel Verwaltungsaufwand benötigt. Die Regeln müssen sicher etwas abgeschwächt werden. Ein vernünftiges Kriterium wurde 1970 von R. Bayer aufgestellt: Jede Seite (ausser einer) enthält zwischen n und 2n Knoten für ein bestimmtes konstantes n [3]. Somit ist in einem Baum mit N Elementen und einer maximalen Seitengrösse von 2n Knoten pro Seite der schlimmste Fall logn(N) Seitenzugriffe - und Seitenzugriffe bestimmen sicher den ganzen Suchaufwand. Ausserdem beträgt der wichtige Faktor der Speicherausnutzung mindestens 50%, da Seiten immer mindestens zur Hälfte gefüllt sind. Neben all diesen Vorteilen enthält das Schema relativ einfache Algorithmen zum Suchen, Einfügen und Löschen. Wir werden sie anschliessend im einzelnen studieren.

Die zugrunde liegenden Datenstrukturen heissen B-Bäume und haben folgende Eigenschaften: n heisst Ordnung des B-Baumes.

1. Jede Seite enthält höchstens 2*n Elemente (Schlüssel).

2. Jede Seite, ausser der Wurzelseite, enthält mindestens n Elemente.

3. Jede Seite ist entweder eine Blattseite, d.h. hat keine Nachfolger, oder sie hat m+1 Nachfolger, wobei m die Zahl ihrer Schlüssel ist.

4. Alle Blattseiten liegen auf der gleichen Stufe.

Fig. 4.45 zeigt einen B-Baum der Ordnung 2 mit drei Stufen. Alle Seiten enthalten 2, 3 oder 4 Elemente; die Ausnahme bildet die Wurzel, die auch nur ein Element enthalten darf. Alle Blattseiten stehen auf Stufe 3. Die Schlüssel erscheinen in aufsteigender Ordnung von links nach rechts, wenn der B-Baum auf eine einzige Stufe zusammengepresst wird, indem man die Nachfolger zwischen die Schlüssel ihrer Vorgängerseite einfügt. Diese Anordnung bildet eine natürliche Erweiterung der Organisation binärer Suchbäume und bestimmt die Art der Suche eines Elementes mit einem gegebenen Schlüssel. Gehen wir aus von einer Seite der in Fig. 4.46 gezeigten Art und einem gegebenen Suchargument x.

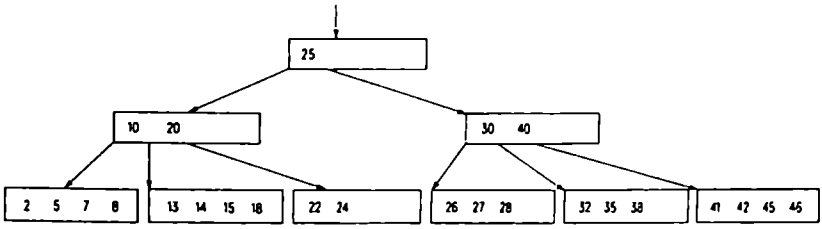


Fig. 4.45: B-Baum zweiter Ordnung

Ist die Seite in den Vordergrundspeicher gebracht, so können wir herkömmliche Methoden zur Suche zwischen den Schlüsseln k1 ... km verwenden. Ist m hinreichend gross, so kann man binäre Suche verwenden, ist es eher klein, genügt eine einfache sequentielle Suche. (Man bedenke, dass die für die Suche im Hauptspeicher notwendige Zeit wahrscheinlich zu vernachlässigen ist im Vergleich zu der Zeit, die benötigt wird, um eine Seite vom Hintergrund- in den Vordergrundspeicher zu bringen.) Hat die Suche keinen Erfolg, so befinden wir uns in einer der folgenden Situationen:

1. k[i] < x < k[i+1] für 1 <= i < m .
   Wir setzen die Suche auf der Seite p[i]↑ fort.

2. k[m] < x . Die Suche wird auf Seite p[m]↑ fortgesetzt.

3. x < k[1] . Die Suche wird auf Seite p[0]↑ fortgesetzt.

Hat der angegebene Zeiger den Wert nil, d.h. gibt es keine nachfolgende Seite, dann existiert in allen drei Fällen im ganzen Baum kein Element mit dem Schlüssel x, und die Suche ist beendet.

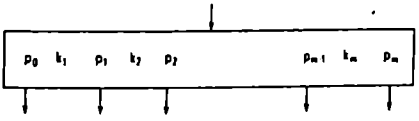


Fig. 4.46: Abschnitt eines B-Baumes mit m Schlüsselwerten

Ueberraschenderweise ist auch das Einfügen in einen B-Baum relativ

einfach. Ist ein Element in eine Seite mit $m < 2n$ Elementen einzufügen, so beschränkt sich der Vorgang des Einfügens auf diese Seite. Nur das Einfügen in eine bereits volle Seite hat Einfluss auf die Seitenstruktur und kann die Zuteilung neuer Seiten veranlassen. Zum Verständnis dieses Vorgangs betrachte man Fig. 4.47, die das Einfügen des Schlüssels 22 in einen B-Baum der Ordnung 2 zeigt. Es erfolgt in drei Schritten:

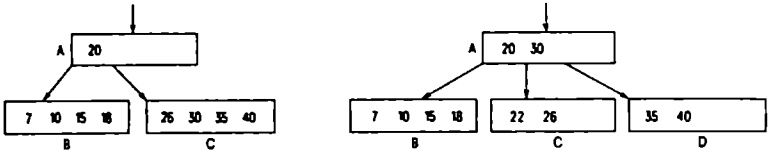


Fig. 4.47: Einfügen des Schlüssels 22

1. Schlüssel 22 fehlt; Einfügen in Seite C ist nicht möglich, da C bereits voll ist.

2. Seite C wird aufgeteilt in zwei Seiten (d.h. eine neue Seite D wird zugeteilt).

3. Die $m+1$ Schlüssel werden gleichmässig auf C und D verteilt, der mittlere Schlüssel wird eine Stufe nach oben in die vorangehende Seite A gebracht.

Dieses sehr elegante Schema bewahrt alle charakteristischen Eigenschaften eines B-Baumes. Insbesondere enthalten die geteilten Seiten genau n Elemente. Natürlich kann das Einfügen eines Elementes in die vorangehende Seite diese wiederum zum Ueberlaufen bringen und somit die Aufteilung fortsetzen. In extremen Fällen kann sie bis zur Wurzel fortgesetzt werden. Dies ist auch der einzige Weg, auf dem die Höhe eines B-Baumes zunehmen kann. Der B-Baum hat somit eine seltsame Art zu wachsen: Er wächst von den Blättern zur Wurzel hinauf.

Wir werden nun aus dieser skizzierenden Beschreibung ein vollständiges Programm entwickeln. Es liegt bereits auf der Hand, dass eine rekursive Formulierung der besonderen Eigenschaft des Aufteilungsprozesses, sich dem Suchpfad entlang nach rückwärts fortzusetzen,

am besten Rechnung trägt. Die allgemeine Struktur des Programms wird daher der des Einfügens in einen ausgeglichenen Baum ähnlich sein, obwohl die Einzelheiten verschieden sind.

Zuerst ist die Struktur der Seite (page) zu definieren. Wir wollen die Elemente in Form eines Array darstellen.

```
type page = record m: index;                                    (4.81)
                   p0: pageref;
                   e: array [1..nn] of item
            end
```

wobei

```
const nn = 2*n;
type pageref = ↑page;
     index = 0..nn
```

und

```
type item = record key: integer;                                (4.82)
                   p: pageref;
                   count: integer
            end
```

Wiederum steht die Komponente "count" von item für irgendeine Art von Information, die jedes Element enthalten kann, die aber beim eigentlichen Suchprozess keine Rolle spielt. Wie man bemerkt, bietet jede Seite Platz für 2n Elemente. Das Feld m gibt an, wieviele Plätze wirklich belegt sind. Da m >= n (ausser für die Wurzelseite), ist eine Speicherausnutzung von mindestens 50 % garantiert.

Der Algorithmus des Durchsuchens und Einfügens bei einem B-Baum ist als Teil von Programm 4.7 in der Prozedur search formuliert. Seine Hauptstruktur liegt auf der Hand und erinnert an die einfache Suche im binären Baum mit dem Unterschied, dass die Entscheidung für die Verzweigung keine binäre Wahl ist. Statt dessen wird die "Suche in einer Seite" als binäre Suche auf dem Array e dargestellt.

Der Algorithmus des Einfügens ist lediglich aus Gründen der Uebersichtlichkeit in einer eigenen Prozedur formuliert. Sie wird aufgerufen, nachdem search ergeben hat, dass ein Element im Baum nach oben (in Richtung der Wurzel) weiterzugeben ist. Dies wird

durch den Booleschen Resultat-Parameter h angezeigt; er spielt eine ähnliche Rolle wie im Algorithmus zum Einfügen in ausgeglichene Bäume, wo h anzeigt, dass der Teilbaum gewachsen ist. Wenn h den Wert true hat, enthält der zweite Resultat-Parameter u das nach oben zu gebende Element. Es ist zu beachten, dass das Einfügen in hypothetischen Seiten beginnt - nämlich in den "speziellen Knoten" von Fig. 4.19; das neue Element wird über den Parameter u sofort zu der Blattseite zum wirklichen Einfügen weitergegeben. Dieses Schema ist in (4.83) skizziert.

```
procedure search(x: integer; a: ref;                    (4.83)
                 var h: Boolean; var u: item);
begin if a = nil then
      begin  {x ist nicht im Baum}
         "Weise x dem Element u zu, setze h auf true, um
          anzuzeigen, dass ein Element nach oben zu geben ist"
      end else
      with a↑ do
      begin {suche x auf Seite a↑}
            "binäres Durchsuchen des Array";
            if "gefunden" then
               "erhöhe den Zähler des
                entsprechenden Elementes"  else
            begin search(x, "Nachfolger", h, u);
               if h then
                  {ein Element u ist nach oben zu geben}
                  if "Zahl der Elemente auf a" < 2*n then
                     "füge u auf Seite a↑ ein und
                      setze h auf false"
                  else "zerlege die Seite und gib mittleres
                        Element nach oben weiter"
            end
      end
end
```

Hat der Parameter h nach dem Aufruf von search im Hauptprogramm den Wert true, so ist die Wurzelseite zu zerlegen. Da die Wurzelseite eine besondere Rolle spielt, ist dieser Vorgang getrennt zu programmieren. Er besteht lediglich aus der Zuteilung einer neuen (Wurzel-) Seite und dem Einfügen des einen durch den Parameter u

gegebenen Elementes. Somit enthält die neue Wurzelseite nur ein einziges Element. Einzelheiten findet man in Programm 4.7.

Fig. 4.48 zeigt das Resultat einer Anwendung von Programm 4.7 zur Konstruktion eines B-Baumes mit der folgenden Eingabe-Sequenz von Schlüsseln:

```
20: 40  10  30  15: 35   7  26  18 / 22:  5:
42  13  46  27   8 \/32: 38  24  45  25:
```

Die Strichpunkte bezeichnen die Positionen, an denen "Schnappschüsse" von der Anordnung der Seiten gemacht wurden. Das Einfügen des letzten Schlüssels verursacht zwei Zerlegungen und die Zuteilung von drei neuen Seiten.

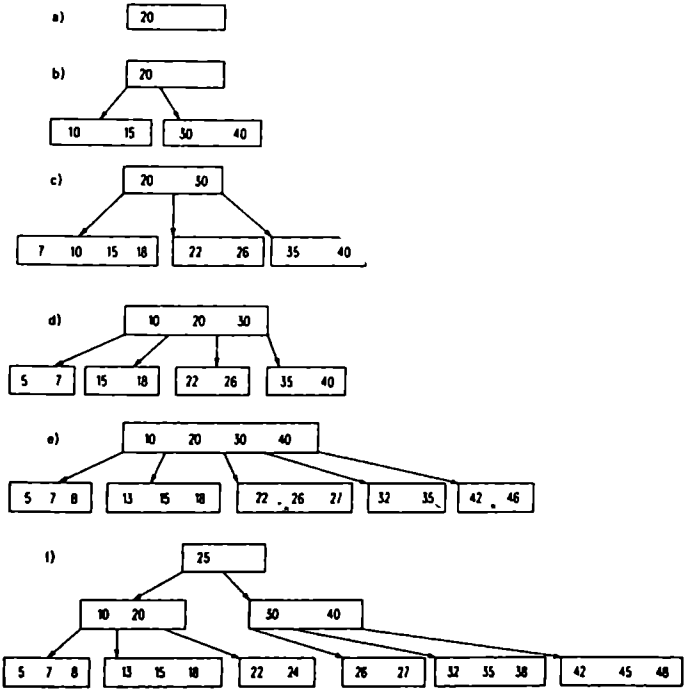


Fig. 4.48: Das Entstehen eines B-Baumes 2. Ordnung

Bemerkenswert ist die spezielle Bedeutung der with-Klausel in diesem Programm, die sich auch schon aus der Skizze (4.83) ergibt. Sie zeigt primär, dass sich die Bezeichner von Seiten-Komponenten in der auf die Klausel folgenden Anweisung automatisch auf die Seite a[beziehen. Sind die Seiten tatsächlich auf einem Hintergrundspeicher - wie es in einem grossen Datenbanksystem sicher notwendig wäre - so könnte die with-Klausel ausserdem so interpretiert werden, dass sie die Uebertragung der angegebenen Seite in den Vordergrundspeicher veranlasst. Da jeder Aufruf von search somit die Zuteilung einer Seite im Hauptspeicher impliziert, sind höchstens k = log$_n$ N rekursive Aufrufe notwendig. Bei einem Baum mit N Elementen müssen wir also in der Lage sein, k Seiten im Hauptspeicher unterzubringen. Dies ist eine Schranke für die Seitengrösse 2*n. Tatsächlich müssen wir sogar mehr als k Seiten unterbringen, da Einfügen die Zerlegung von Seiten verursachen kann. Zudem sollte die Wurzelseite am besten dauernd im Vordergrundspeicher sein, da jede Anfrage notwendigerweise von der Wurzelseite ausgeht.

Eine andere positive Eigenschaft der Organisation des B-Baumes ist seine Eignung und Wirtschaftlichkeit für das rein sequentielle Abändern der ganzen Datenbank. Jede Seite wird genau einmal in den Vordergrundspeicher gebracht.

Löschen von Elementen eines B-Baumes ist im Prinzip naheliegend, im einzelnen jedoch ziemlich kompliziert. Wir können zwei Fälle unterscheiden:

1. Das zu löschende Element ist auf einer Blattseite; dann ist der Lösch-Algorithmus klar und einfach.

2. Das Element ist nicht auf einer Blattseite; in diesem Fall ist es durch eines der lexikographisch benachbarten Elemente zu ersetzen, die auf Blattseiten sein können und dann leicht zu löschen sind.

Im Fall 2 ist die Bestimmung des benachbarten Schlüssels analog der beim Löschen im binären Baum. Entlang den am weitesten rechts stehenden Zeigern steigen wir hinab zu der Blattseite P, ersetzen das zu löschende Element durch das äusserst rechte Element von P

und verkleinern dann die Grösse von P um 1.

Auf jeden Fall muss nach der Reduktion der Grösse die Zahl m der Elemente auf der verkleinerten Seite geprüft werden. Denn m < n wäre eine Verletzung der wesentlichen Eigenschaft der B-Bäume. Eine zusätzliche Aktion ist durchzuführen; diese Unterlauf-Bedingung wird durch den Booleschen Variablen-Parameter h angegeben.

Abhilfe ist nur zu schaffen durch Ausleihen oder "Angliedern" eines Elementes von einer der benachbarten Seiten. Da dazu diese Seite Q in den Hauptspeicher gebracht werden muss - eine relativ kostspielige Operation - ist man versucht, durch Angliedern von mehr als einem Element das beste aus dieser unerwünschten Situation zu machen. Nach der üblichen Strategie werden die Elemente der Seiten P und Q gleichmässig auf beide Seiten verteilt. Dieser Vorgang heisst Ausgleichen.

Es kann natürlich vorkommen, dass kein Element zum Angliedern übrig ist, da Q bereits die minimale Grösse n erreicht hat. In diesem Fall enthalten die Seiten P und Q zusammen 2n-1 Elemente; wir können die beiden Seiten zu einer einzigen zusammenlegen, indem wir das mittlere Element der P und Q vorangehenden Seite hinzufügen und die Seite Q ganz wegnehmen. Dies ist genau der zur Aufteilung einer Seite inverse Vorgang. Man kann sich den Prozess vorstellen, wenn man Schlüssel 22 in Fig. 4.47 löscht.

Das Herausnehmen des mittleren Schlüssels in der vorangehenden Seite kann noch einmal die Grösse unter die erlaubte Grenze n fallen lassen und somit auf der nächsten Stufe eine weitere spezielle Aktion (Ausgleichen oder Mischen) hervorrufen. Im extremen Fall kann sich das Mischen von Seiten bis zur Wurzel nach oben fortsetzen. Reduziert sich die Grösse der Wurzel zu 0, wird sie selbst gelöscht und verursacht somit eine Reduktion der Höhe des B-Baumes. Dies ist sogar die einzige Art, auf die ein B-Baum bezüglich seiner Höhe schrumpfen kann.

Fig. 4.49 zeigt das schrittweise Abnehmen des B-Baumes von Fig. 4.48 durch aufeinanderfolgendes Löschen der Schlüssel

25 45 24; 38 32; 8 27 46 13 42;
5 22 18 26; 7 35 15;

Die Strichpunkte bezeichnen wiederum die Stellen, an denen "Schnappschüsse" gemacht und auch Seiten weggenommen werden. Der Algorithmus des Löschens ist als Prozedur in Programm 4.7 enthalten. Bemerkenswert ist ganz besonders die Aehnlichkeit seiner Struktur mit der des Löschens im ausgeglichenen Baum.

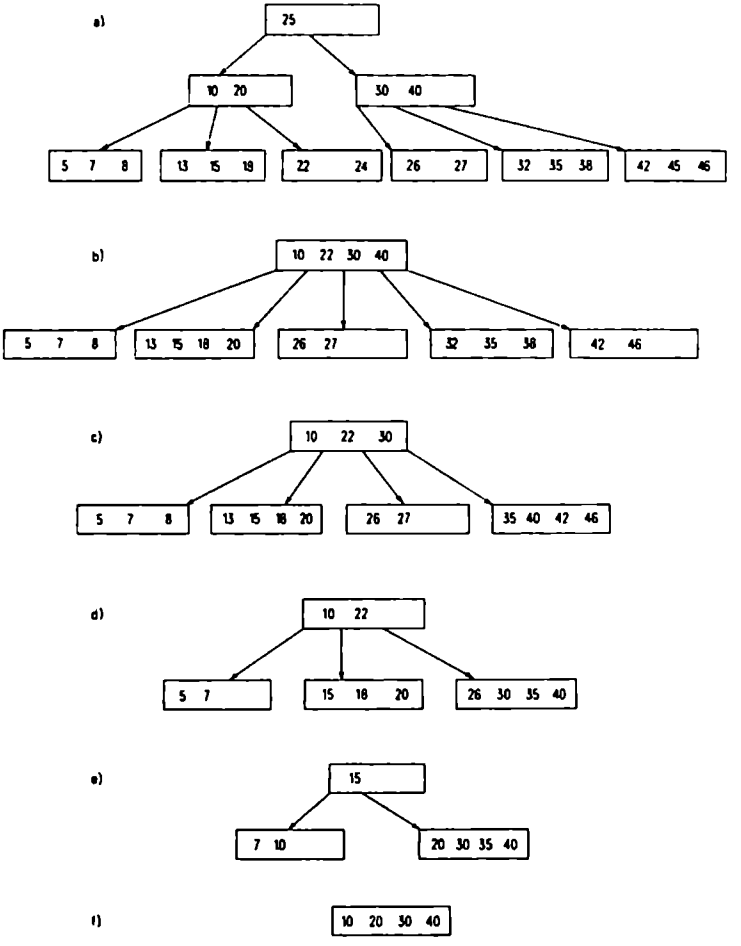


Fig. 4.49: Zerfall eines B-Baumes 2. Ordnung

```
program Btree(input,output);
{Durchsuchen, Einfügen und Löschen im B-Baum}
const n =2; nn = 4;  {Seitengrösse}
type ref = ↑page;
   item = record key: integer;
             p: ref;
             count: integer;
          end ;
   page = record m: 0..nn;  {Anzahl von Elementen}
             p0: ref;
             e : array [1..nn] of item;
          end ;
var root, q: ref; x: integer;
   h: boolean;  u: item;

procedure printtree(p: ref; l: integer);
   var i: integer;
begin if p <> nil then
   with p↑ do
   begin for i := 1 to l do write('    ');
         for i := 1 to m do write(e[i].key: 4);
         writeln;
         printtree(p0,l+1);
         for i := 1 to m do printtree(e[i].p, l+1)
   end
end ;

procedure search(x: integer; a:ref;
                 var h: boolean; var v: item);
{Suche Schlüssel x im B-Baum mit Wurzel a; falls vorhanden
 erhöhe den Zähler, sonst füge ein Element mit Schlüssel x
 und Zähler 1 in den Baum ein. Ein Element, das auf eine
 tiefere Stufe zu bringen ist, ist v zuzuweisen;
 h := 'Baum ist gewachsen'}

   var k,l,r: integer; q: ref; u: item;

   procedure insert;
      var i: integer; b: ref;
   begin {füge u rechts von a↑.e[r] ein}
```

```
      with a↑ do
      begin if m < n then
         begin m := m+1; h := false;
            for i := m downto r+2 do e[i] := e[i-1];
            e[r+1] := u
         end else
         begin {Seite a ist voll; teile sie auf und
                weise v das herauskommende Element zu}
            new(b);
            if r <= n then
            begin if r = n then v := u else
                  begin v := e[n];
                     for i := n downto r+2 do e[i] := e[i-1];
                     e[r+1] := u
                  end ;
               for i := 1 to n do b↑.e[i] := a↑.e[i+n];
            end else
            begin {füge u in die rechte Seite ein}
               r := r-n; v := e[n+1];
               for i := 1 to r-1 do
                  b↑.e[i] := a↑]e[i+n+1];
               b↑.e[r] := u;
               for i := r+1 to n do b↑.e[i] := a↑.e[i+n]
            end ;
            m := n; b↑.m := n; b↑.p0 := v.p; v.p := b;
         end
      end {with}
   end {insert} ;

begin {suche Schlüssel x auf Seite a↑; h = false}
   if a = nil then
   begin {Element mit Schlüssel x ist nicht im Baum}
      h := true;
      with v do
      begin key := x; count := 1; p := nil
      end
   end else
   with a↑ do
   begin l := 1; r := m;  {binäres Durchsuchen des Array}
      repeat k := (l+r) div 2;
```

```
            if x <= e[k].key then r := k-1;
            if x >= e[k].key then l := k+1;
         until r < l;
         if l-r > 1 then
         begin {gefunden}
            e[k].count := e[k].count + 1;  h:= false
         end else
         begin {Element ist nicht in dieser Seite}
            if r = 0 then q := p0 else q := e[r].p;
            search(x,q,h,u); if h then insert
         end
      end
   end {search} ;

   procedure delete(x: integer; a: ref; var h: boolean);
   {Suche und lösche Schlüssel x im B-Baum a; tritt Unterlauf
    ein, so gleiche falls möglich mit benachbarter
    Seite aus, sonst mische;
    h:= 'Seite ist nicht voll genug'}

      var i,k,l,r: integer; q: ref;

      procedure underflow(c,a: ref; s: integer; var h: boolean);
         {a = Seite mit Unterlauf, c = Vorgänger-Seite}
         var b: ref; i,k,mb,mc: integer;
      begin mc := c↑.m;  {h = true, a↑.m = n-1}
         if s < mc then
         begin {b := rechts von a}  s := s+1;
            b := c↑.e[s].p; mb := b↑.m; k := (mb-n+1) div 2;
            {k = Anzahl Elemente auf benachbarter Seite b}
            a↑.e[n] := c↑.e[s]; a↑.e[n].p := b↑.p0;
            if k > 0 then
            begin {bringe k Elemente von b nach a}
               for i := 1 to k-1 do a↑.e[i+n] := b↑.e[i];
               c↑.e[s] := b↑.e[k]; c↑.e[s].p := b;
               b↑.p0 := b↑.e[k].p; mb := mb-k;
               for i := 1 to mb do b↑.e[i] := b↑.e[i+k];
               b↑.m := mb; a↑.m := n-1+k; h := false
            end else
            begin {mische Seiten a und b}
```

```
            for i := 1 to n do a↑.e[i+n] := b↑.e[i];
            for i := s to mc-1 do c↑.e[i] := c↑.e[i+1];
            a↑.m := nn; c↑.m := mc-1; {dispose(b)}
         end
      end else
      begin {b := Seite links von a}
         if s = 1 then b := c↑.p0 else b := c↑.e[s-1].p;
         mb := b↑.m + 1; k := (mb-n) div 2;
         if k > 0 then
         begin {bringe k Elemente von b nach a}
            for i := n-1 downto 1 do a↑.e[i+k] := a↑.e[i];
            a↑.e[k] := c↑.e[s]; a↑.e[k].p := a↑.p0;
            mb := mb-k;
            for i := k-1 downto 1 do a↑.e[i] := b↑.e[i+mb];
            a↑.p0 := b↑.e[mb].p;
            c↑.e[s] := b↑.e[mb]; c↑.e[s].p := a;
            b↑.m := mb-1; a↑.m := n-1+k; h := false
         end else
         begin {mische Seiten a und b}
            b↑.e[mb] := c↑.e[s]; b↑.e[mb].p := a↑.p0;
            for i := 1 to n-1 do b↑.e[i+mb] := a↑.e[i];
            b↑.m := nn; c↑.m := mc-1; {dispose(a)}
         end
      end
   end {underflow} ;

   procedure del(p: ref; var h: boolean);
      var q: ref;    {a, k global}
   begin
      with p↑ do
      begin q := e[m].p;
         if q <> nil then
         begin del(q,h); if h then underflow(p,q,m,h)
         end else
         begin p↑.e[m].p := a↑.e[k].p; a↑.e[k] := p↑.e[m];
            m := m-1; h := m<n
         end
      end
   end {del} ;
```

```
begin {delete}
   if a = nil then
   begin writeln(' key is not in tree'); h := false
   end else
   with a↑ do
   begin l := 1; r := m;  {binäres Durchsuchen des Array}
      repeat k := (l+r) div 2;
         if x <= e[k].key then r := k-1;
         if x >= e[k].key then l := k+1;
      until l > r;
      if r=0 then q := p0 else q := e[r].p;
      if l-r > 1 then
      begin {gefunden; lösche e[k]}
         if q = nil then
         begin {a ist eine Endseite}  m := m-1; h := m<n;
            for i := k to m do e[i] := e[i+1];
         end else
         begin del(q,h); if h then underflow(a,q,r,h)
         end
      end else
      begin delete(x,q,h); if h then underflow(a,q,r,h)
      end
   end
end {delete} ;

begin root := nil; read(x);
   while x <> 0 do
   begin writeln(' search key', x);
      search(x,root,h,u);
      if h then
      begin {füge neue Wurzelseite ein} q := root; new(root);
         with root↑ do
            begin m := 1; p0 := q; e[1] := u
            end
      end ;
      printtree(root,1); read(x)
   end ;
   read(x);
   while x <> 0 do
   begin writeln(' delete key', x);
```

```
        delete(x,root,h);
        if h then
        begin {Grösse der Wurzelseite wurde reduziert}
           if root↑.m = 0 then
           begin q := root; root := q↑.p0; {dispose(q)}
           end ;
        end ;
        printtree(root,1); read(x)
     end
end .
```

Programm 4.7

Eine ausführliche Analyse der Leistung von B-Bäumen wurde von Bayer und McCreight erstellt [3]. Sie behandelt besonders die Frage der optimalen Grösse n der Seite, die stark von den Eigenschaften des zur Verfügung stehenden Speichers und Rechensystems abhängt.

Varianten des Schemas des B-Baumes werden in [18] erörtert. Eine wichtige Beobachtung ist, dass das Aufteilen sowie das Zusammenlegen von Seiten verschoben werden sollte, indem man zunächst versucht, benachbarte Seiten auszugleichen. Abgesehen davon, scheinen die vorgeschlagenen Verbesserungen nur unbedeutende Gewinne zu erzielen.

4.5.2. Binäre B-Bäume

Die Art von B-Bäumen, die am uninteressantesten zu sein scheint, sind die B-Bäume erster Ordnung (n = 1). Manchmal lohnt es sich jedoch, auch diese Fälle zu betrachten. Es ist klar, dass B-Bäume erster Ordnung zur Darstellung grosser geordneter und indizierter Datenmengen unter Verwendung von Hintergrundspeicher nicht sinnvoll sind. Denn ungefähr 50% aller Seiten werden nur ein Element enthalten. Wir werden deshalb Hintergrundspeicher beiseite lassen und wieder das Problem des Suchens in Bäumen auf nur einem Speicherniveau betrachten.

Ein binärer B-Baum (BB-Baum) besteht aus Knoten (Seiten) mit

entweder einem oder zwei Elementen. Eine Seite enthält daher zwei oder drei Zeiger zu Nachfolgern; dies führt zur Bezeichnung 2-3-Baum. Entsprechend der Definition von B-Bäumen sind alle Blattseiten auf der gleichen Stufe und haben alle Nicht-Blattseiten von BB-Bäumen zwei oder drei Nachfolger (inklusive Wurzel). Da wir es nun nur mit Vordergrundspeicher zu tun haben, ist eine optimale Speicherausnutzung notwendig, und die Darstellung der Elemente innerhalb eines Knotens durch einen Array erscheint ungeeignet. Eine Alternative bietet die dynamisch verkettete Zuteilung; d.h. innerhalb eines jeden Knotens gibt es eine verkettete Liste von Elementen mit der Länge 1 oder 2. Da jeder Knoten höchstens drei Nachfolger hat und somit nur bis zu drei Zeiger unterbringen muss, ist man versucht, die Zeiger für die Nachfolger mit den Zeigern der Elementenliste wie in Fig. 4.50 zusammenzufassen. Die Knoten des B-Baumes verlieren dabei ihre eigentliche Identität, und die Elemente übernehmen die Rolle eines Knotens in einem regulären binären Baum. Es ist aber notwendig, zwischen Zeigern zu Nachfolgern (vertikal) und solchen zu "Geschwistern" auf derselben Seite (horizontal) zu unterscheiden. Da nur Zeiger nach rechts horizontal sein können, genügt ein Bit zur Unterscheidung.

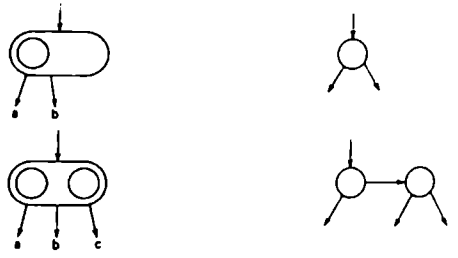


Fig. 4.50: Darstellung von BB-Baumelementen

Wir führen daher das Boolesche Feld h mit der Bedeutung "horizontal" ein. (4.84) enthält die Definition eines auf dieser Darstellung beruhenden Knotens. Sie wurde von R. Bayer 1971 vorgeschlagen und untersucht [2] und stellt die Organisation eines Suchbaumes mit garantierter maximaler Weglänge p = 2*round(log(N)+0.5) dar.

```
type node = record key: integer;                    (4.84)
                   ...
                   left,right: ref;
                   h: Boolean
            end
```

Beim Problem des Einfügens eines Schlüssels sind vier mögliche Situationen zu unterscheiden, die sich aus dem Wachsen des linken oder rechten Teilbaumes ergeben. Fig. 4.51 illustriert diese vier Fälle. Es wird in Erinnerung gerufen, dass B-Bäume die Eigenschaft haben, von unten zur Wurzel hinaufzuwachsen, und dass alle Blätter auf der gleichen Stufe bleiben müssen.

Im einfachsten Fall (1) wächst der rechte Teilbaum eines Knotens A, der der einzige Schlüssel auf seiner (hypothetischen) Seite ist.

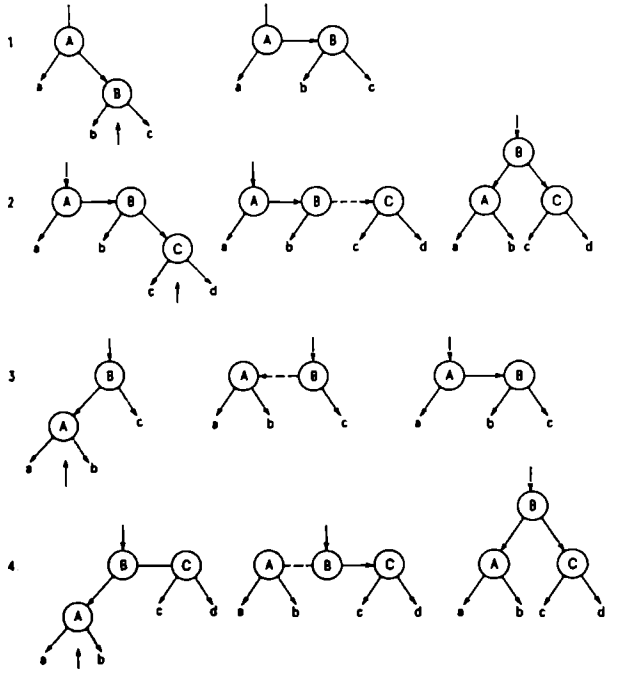


Fig. 4.51: Einfügen von Elementen in BB-Bäumen

Dann wird der Nachfolger B lediglich Geschwisterteil von A, d.h. der vertikale Zeiger wird horizontal. Dieses einfache "Hochnehmen" des rechten Armes ist nicht möglich, wenn A bereits einen Geschwisterteil hat. Dann erhalten wir eine Seite mit drei Knoten und müssen sie aufteilen (Fall 2). Ihr mittlerer Knoten B wird auf die nächst höhere Stufe gebracht.

Nun nehmen wir an, dass die Höhe des linken Teilbaumes eines Knotens B zugenommen hat. Ist B wieder allein auf einer Seite (Fall 3), d.h. zeigt sein rechter Zeiger zu einem Nachfolger, so kann der linke Teilbaum (A) ein Geschwisterteil von B werden. (Eine einfache Rotation der Zeiger ist dazu notwendig, da der linke Zeiger nicht horizontal sein kann.) Hat B aber bereits einen Geschwisterteil, so führt das Hochnehmen von A zu einer Seite mit drei Elementen und erfordert ein Aufteilen. Diese Teilung erfolgt auf einfache Art: C wird Nachfolger von B, das zur nächst höheren Stufe aufsteigt (Fall 4).

Es ist zu beachten, dass beim Suchen eines Schlüssels kein wirklicher Unterschied besteht, ob wir einem horizontalen oder vertikalen Zeiger folgen. Es erscheint deshalb gekünstelt, wenn man darüber besorgt ist, dass der linke Zeiger in Fall 3 horizontal wird, obwohl die Seite immer noch nicht mehr als zwei Elemente enthält. In der Tat zeigt der Algorithmus des Einfügens eine seltsame Asymmetrie in der Behandlung des Wachsens des linken und rechten Teilbaumes und lässt die Organisation des BB-Baumes ziemlich unnatürlich erscheinen. Es gibt keinen "Beweis" für die Eigenartigkeit dieser Organisation, aber eine gesunde Intuition sagt uns, dass etwas "faul" ist und wir diese Asymmetrie beseitigen sollten. Dies führt zum Begriff des symmetrischen binären Baumes (SBB-Baum), der auch von Bayer [4] 1972 untersucht wurde. Er führt zu im Mittel leicht effizienteren Suchbäumen, aber die Algorithmen für Einfügen und Löschen sind etwas komplexer. Ausserdem benötigt jeder Knoten nun zwei Bits (Boolesche Variable lh und rh) zur Angabe der Art seiner beiden Zeiger.

Wir beschränken unsere Einzelbetrachtungen auf das Problem des Einfügens und haben wiederum zwischen vier Fällen des Wachstums von Teilbäumen zu unterscheiden. Sie sind in Fig. 4.52 dargestellt, die die erreichte Symmetrie gut zum Ausdruck bringt. Jedesmal wenn ein

Teilbaum eines Knotens A ohne Geschwister wächst, wird die Wurzel des Teilbaumes zum Geschwisterteil von A. Auf diesen Fall wollen wir nicht weiter eingehen.

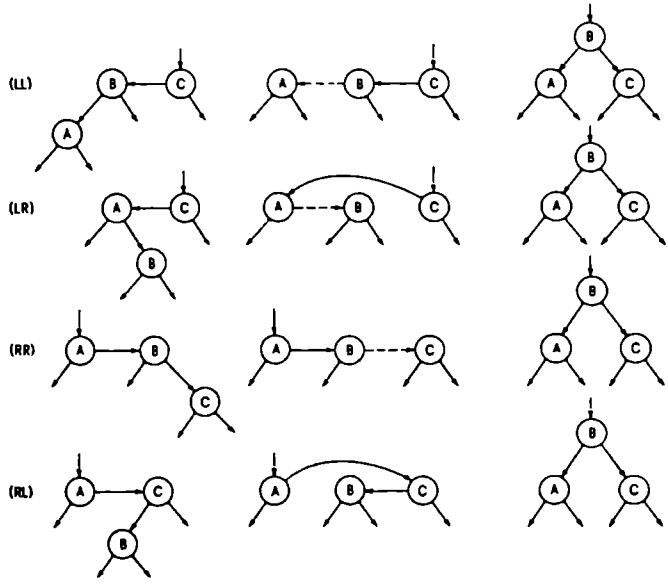


Fig. 4.52: Einfügen von Elementen im SBB-Baum

Alle vier in Fig. 4.52 betrachteten Fälle zeigen das Auftreten eines SeitenÜberlaufs und das anschliessende Aufteilen der Seite. Sie sind je nach den Richtungen der horizontalen Zeiger bezeichnet, die die drei Geschwister in den mittleren Abbildungen verknüpfen. Die linke Kolonne zeigt die Ausgangssituation, die mittlere Kolonne die Tatsache, dass der tiefere Knoten durch das Wachsen seines Teilbaumes aufgestiegen ist, und die Abbildungen der rechten Kolonne illustrieren die neue Anordnung der Knoten ("Aufteilung der Seite").

Es ist nicht ratsam, länger am Begriff der Seiten festzuhalten, aus dem diese Organisation entwickelt wurde, denn wir sind nur daran interessiert, die maximale Weglänge auf 2*log(N) zu beschränken. Dazu müssen wir nur sicherstellen, dass auf keinem Suchpfad zwei horizontale Zeiger aufeinanderfolgen. Es gibt jedoch keinen Grund,

Knoten mit horizontalen Zeigern nach links und rechts zu verbieten. Wir werden deshalb die SBB-Bäume als Bäume mit folgenden Eigenschaften definieren:

1. Jeder Knoten enthält einen Schlüssel und höchstens zwei (Zeiger zu) Teilbäume(n).

2. Jeder Zeiger ist entweder horizontal oder vertikal. Auf keinem Suchpfad gibt es zwei aufeinanderfolgende horizontale Zeiger.

3. Alle Endknoten (Knoten ohne Nachfolger) erscheinen auf der gleichen (End-) Stufe.

Aus dieser Definition folgt, dass der längste Suchweg nicht länger ist als die doppelte Höhe des Baumes. Da kein SBB-Baum mit N Knoten höher sein kann als round(log(N)+0.5), ergibt sich unmittelbar 2*round(log(N)+0.5) als obere Schranke für die Länge des Suchpfades.

Damit der Leser sich vorstellen kann, wie diese Bäume wachsen, sei er auf Fig. 4.53 verwiesen. Die Zeilen stellen Schnappschüsse dar, die während des Einfügens der folgenden Sequenzen von Schlüsseln beim Auftreten eines Strichpunkts gemacht wurden.

```
(1)   1  2; 3; 4  5  6; 7;                    (4.85)
(2)   5  4; 3; 1  2  7  6;
(3)   6  2; 4; 1  7  3  5;
(4)   4  2  6; 1  7; 3  5;
```

Diese Bilder heben besonders die dritte Eigenschaft der B-Bäume hervor: Alle Endknoten erscheinen auf der gleichen Stufe. Man ist deshalb versucht, diese Strukturen mit frisch geschnittenen Gartenhecken zu vergleichen und sie Hecken zu nennen.

Der Algorithmus für die Konstruktion von Hecken-Bäumen ist in (4.87) formuliert. Er stützt sich auf die Definition (4.86) des Typs node mit den zwei Komponenten lh und rh, die die horizontale Lage des linken und rechten Zeigers angegeben.

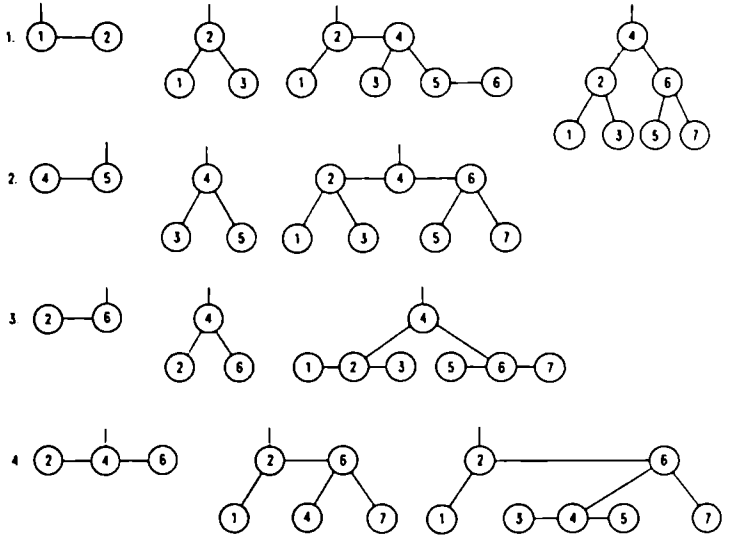


Fig. 4.53: Das Enstehen von "Hecken" mit Eingabefolge (4.85)

```
type node = record key: integer;                    (4.86)
                   count: integer;
                   left,right: ref;
                   lh,rh: Boolean
            end
```

Die rekursive Prozedur search folgt wiederum dem Muster des grundlegenden Algorithmus zum Einfügen in binäre Bäume (siehe (4.87)). Ein dritter Parameter h wird hinzugefügt, der angibt, ob sich der Teilbaum mit Wurzel p verändert hat, und der direkt dem Parameter h des Suchprogramms für binäre Bäume entspricht. Wir müssen aber eine Konsequenz aus der Darstellung von "Seiten" als verkettete Listen ziehen: Eine Seite wird durch entweder einen oder zwei Aufrufe der Prozedur search durchlaufen. Wir müssen unterscheiden zwischen dem Fall eines grösser gewordenen Teilbaumes (bezeichnet durch einen vertikalen Zeiger) und einem Geschwister-Knoten (bezeichnet durch einen horizontalen Zeiger), der einen weiteren Geschwisterteil erhielt und somit das Aufteilen einer Seite erforderlich macht. Das Problem ist leicht zu lösen durch

Einführung eines drei-wertigen h mit folgender Bedeutung:

h = 0: Der Teilbaum p erfordert kein Verändern der Baumstruktur.
h = 1: Der Knoten p hat einen Geschwisterteil erhalten.
h = 2: Die Grösse des Teilbaumes p hat zugenommen.

```
procedure search(x: integer; var p: ref; var h: integer);
   var p1,p2: ref;
begin                                                    (4.87)
   if p = nil then
   begin {Wort ist nicht im Baum; füge es ein}
      new(p); h := 2;
      with p↑ do
      begin key := x; count := 1; left := nil;
         right := nil; lh := false; rh := false
      end
   end else
   if x < p↑.key then
   begin search(x,p↑.left,h);
      if h <> 0 then
      if p↑.lh then
      begin p1 := p↑.left; h := 2; p↑.lh := false;
         if p1↑.lh then
         begin {LL} p↑.left := p1↑.right;
            p1↑.right := p; p1↑.lh := false; p := p1
         end else
         if p1↑.rh then
         begin {LR} p2 := p1↑.right; p1↑.rh := false;
            p1↑.right := p2↑.left; p2↑.left := p1;
            p↑.left := p2↑.right; p2↑.right := p; p := p2
         end
      end else
      begin h := h-1; if h <> 0 then p↑.lh := true
      end
   end else
   if x > p↑.key then
   begin search(x,p↑.right,h);
      if h <> 0 then
      if p↑.rh then
      begin p1 := p↑.right; h := 2; p↑.rh := false;
```

```
            if p1↑.rh then
            begin {RR} p↑.right := p1↑.left;
               p1↑.left := p; p1↑.rh := false; p := p1
            end else
            if p1↑.lh then
            begin {RL} p2 := p1↑.left; p1↑.lh := false;
               p1↑.left := p2↑.right; p2↑.right := p1;
               p↑.right := p2↑.left; p2↑.left := p; p := p2
            end
         end else
         begin h := h-1; if h <> 0 then p↑.rh := true
         end
      end else
      begin p↑.count := p↑.count + 1; h := 0
      end
   end {search}
```

Bemerkenswert ist die Aehnlichkeit der für die Neuordnung der Knoten notwendigen Aktionen mit denjenigen, die im Such-Algorithmus (4.63) für ausgeglichene Bäume entwickelt wurden. Nach Programm 4.8 können alle vier Fälle offensichtlich durch einfache Zeigerrotationen implementiert werden, durch einzelne Rotationen in den Fällen LL und RR und durch doppelte Rotationen in den Fällen LR und RL. Die Prozedur (4.87) ist etwas einfacher als (4.63). Das Schema des Hecken-Baumes bildet eine Alternative zum Kriterium der AVL-Ausgeglichenheit. Ein Vergleich der Leistungen ist daher möglich und wünschenswert.

Wir sehen von einer mathematischen Analyse ab und beschränken uns auf einige grundlegende Unterschiede. Es kann bewiesen werden, dass AVL-ausgeglichene Bäume eine Teilmenge der Hecken-Bäume sind. Die Klasse der letzteren ist daher umfassender. Es folgt, dass ihre Weglänge im Mittel grösser ist als im AVL Fall. In diesem Zusammenhang ist der Baum (4) "des schlimmsten Falls" in Fig. 4.53 zu beachten. Anderseits wird das Neuordnen der Knoten weniger oft vorkommen. Der ausgeglichene Baum ist daher bei solchen Anwendungen vorzuziehen, bei denen das Wiederauffinden von Schlüsseln häufiger ist als Einfügen (oder Löschen); ist dieses Verhältnis klein, so kann man dem Schema des Hecken-Baumes den Vorzug geben.

Es ist sehr schwierig, eine Grenze zu ziehen. Sie hängt nicht nur sehr stark vom Quotient zwischen Häufigkeit des Wiederauffindens und Veränderns der Struktur ab, sondern auch von den Eigenschaften seiner Implementation. Dies ist besonders dann der Fall, wenn die Knoten-Records dicht gepackt dargestellt werden und der Zugriff zu Feldern somit das Auspacken eines Wortteils bedingt. Boolesche Felder (lh, rh im Fall von Hecken-Bäumen) können bei den meisten Implementationen effizienter behandelt werden als drei-wertige Felder (bal im Fall ausgeglichener Bäume).

4.6. Schlüssel-Transformationen (hashing)

Folgendes allgemeine Problem wurde im letzten Abschnitt behandelt und zur Entwicklung von Lösungen verwendet, die die Techniken der dynamischen Speicherverwaltung zeigten:

> Gegeben sei eine Menge M von Elementen, die durch Schlüsselwerte charakterisiert sind. Auf M ist eine Ordnungsrelation definiert. Wie ist M zu repräsentieren, damit ein Element mit gegebenem Schlüssel k mit kleinstmöglichem Aufwand wiedergefunden werden kann?

Im Speicher einer Rechenanlage wird schliesslich jedes Element durch eine Speicheradresse a bezeichnet. Folglich besteht das gestellte Problem im wesentlichen aus dem Finden einer Abbildung H von Schlüsseln (K) auf Adressen (A):

$$H: K \rightarrow A$$

In Abschnitt 4.5 wurde diese Abbildung in Form von verschiedenen Listen- und Baum-Such-Algorithmen mit unterschiedlichen zugrunde liegenden Datenorganisationen implementiert. Hier zeigen wir noch einen anderen Zugang, der in vielen Fällen sehr effizient und im Grunde einfach ist. Dass er auch einige Nachteile hat, wird später ersichtlich.

Die bei dieser Technik verwendete Datenorganisation ist die Array-

Struktur. H bildet demnach Schlüssel auf Array-Indizes ab, und dies ist auch der Grund für den Begriff Schlüssel-Transformation, der allgemein für diese Technik verwendet wird. Wir müssen uns nicht auf irgendwelche dynamische Zuweisungsprozeduren beziehen, da der Array eine der grundlegenden statischen Strukturen ist. Dieser Abschnitt ist somit unter der Kapitelüberschrift dynamische Informationsstrukturen eigentlich fehl am Platz, aber da Schlüssel-Transformationen oft in Problembereichen verwendet werden, in denen Baumstrukturen vergleichbare Mitbewerber sind, scheint dies doch ein geeigneter Platz für ihre Darstellung zu sein.

Die Hauptschwierigkeit bei der Verwendung einer Schlüssel-Transformation liegt darin, dass die Menge der möglichen Schlüsselwerte sehr viel grösser ist als die Menge der verfügbaren Speicheradressen (Array-Indizes). Ein typisches Beispiel ist die Verwendung alphabetischer Worte mit etwa bis zu 10 Buchstaben als Schlüssel für die Bezeichnung einzelner in einer Menge von z.B. bis zu tausend Personen. Hier gibt es 26**10 mögliche Schlüssel, die auf 1000 mögliche Indizes abzubilden sind. Die Funktion H ist also offensichtlich eine Funktion, die mehrere Argumente auf einen Wert abbildet. Für einen gegebenen Schlüssel k ist der erste Schritt beim Wiederauffinden (Suchen) die Berechnung des zugewiesenen Index h = H(k), und der zweite - daher sicher notwendige - Schritt ist die Prüfung, ob das Element mit dem Schlüssel k tatsächlich durch h im Array (Tabelle) T bezeichnet wird, d.h. ob T[H(k)].key = k. Es stellen sich sofort zwei Fragen:

1. Welche Art von Funktion H sollte verwendet werden?
2. Wie behandeln wir den Fall, dass H den Ort des gewünschten Elementes nicht liefert?

Die Antwort auf Frage 2 ist die Verwendung einer Methode zur Berechnung eines alternativen Platzes, sagen wir Index h', und, ist dies immer noch nicht der Ort des gesuchten Elementes, eines dritten Index h'', usw. Die Situation, in der ein anderer als der gewünschte Schlüssel an der bezeichneten Stelle ist, heisst eine Kollision, und die Aufgabe, alternative Indizes zu erzeugen, wird Behandlung der Kollision genannt. Im folgenden wollen wir die Wahl einer Transformationsfunktion und Methoden zur Behandlung der Kollision erörtern.

4.6.1. Wahl einer Transformationsfunktion H

Eine Voraussetzung für eine gute Transformationsfunktion ist eine möglichst gleichmässige Verteilung der Schlüssel auf den Bereich der Indexwerte. Neben dieser Bedingung ist die Verteilung an kein Muster gebunden, und es ist geradezu wünschenswert, wenn sie völlig zufällig zu sein scheint. Diese Eigenschaft hat der Methode den nicht sehr wissenschaftlich klingenden Namen hashing eingebracht, d.h. "Zerhacken des Argumentes" oder "einen Mischmasch machen". Sie ist auch unter dem Namen Streuspeicherung bekannt. H heisst hash-Funktion und sollte effizient berechenbar sein, d.h. aus sehr wenigen arithmetischen Grundoperationen bestehen.

Man nehme an, dass eine Umwandlungsfunktion ord(k) zur Verfügung steht und die Ordinalzahl des Schlüssels k in der Menge aller möglichen Schlüssel bezeichnet. Weiter nehme man an, dass die Array-Indizes i sich über den Bereich der ganzen Zahlen Ø...N-1 erstrecken, wobei N die Grösse des Array ist. Dann ist eine einfache Wahl

$$H(k) = ord(k) \ \underline{mod} \ N \qquad (4.88)$$

Die Funktion hat die Eigenschaft, die Schlüsselwerte gleichmässig über den Indexbereich zu streuen, und ist daher Grundlage der meisten Schlüsseltransformationen. Sie ist auch sehr einfach zu berechnen, wenn n eine Potenz von 2 ist. Aber genau dies ist zu vermeiden, wenn die Schlüssel Buchstabenfolgen sind. Denn in diesem Fall ist die Annahme der Gleichverteilung der Schlüssel falsch. Tatsächlich werden Worte, die sich nur in wenigen Zeichen am Anfang unterscheiden, mit grosser Wahrscheinlichkeit auf gleiche Indizes abgebildet und können somit zu einer höchst ungleichmässigen Verteilung führen. In (4.88) ist es deshalb besonders empfehlenswert, für N eine Primzahl zu wählen [20]. Dies hat zur Folge, dass eine vollständige Divisionsoperation durchzuführen ist, die nicht durch reines Herausgreifen binärer Ziffern ersetzt werden kann. Dies ist jedoch bei den meisten modernen Rechenanlagen kein ernsthafter Nachteil, da sie über eine eingebaute Divisionsinstruktion verfügen.

Oft werden hash-Funktionenen verwendet, die logische Operationen

wie das "exklusive oder" auf einige Teile des als Sequenz binärer Ziffern dargestellten Schlüssels anwenden. Diese Operationen mögen auf manchen Rechenanlagen etwas schneller sein als die Division, versagen manchmal aber auffallend bei der gleichmässigen Verteilung der Schlüssel auf den Bereich der Indizes. Wir sehen daher von der Erörterung solcher Methoden in weiteren Einzelheiten ab.

4.6.2. Behandlung der Kollision

Wenn sich eine einem gegebenen Schlüssel entsprechende Eintragung in einer Tabelle nicht als das gewünschte Element erweist, liegt eine Kollision vor, d.h. zwei Elemente besitzen Schlüssel, die auf den gleichen Index abgebildet werden. Ein zweite Sondierung ist notwendig unter Verwendung eines Index, der sich auf eine deterministische Art aus dem gegebenen Schlüssel ergibt. Es gibt mehrere Mehoden zur Erzeugung sekundärer Indizes. Naheliegend und wirkungsvoll ist die Verknüpfung aller Eintragungen mit identischem primärem Index H(k) in einer Liste, die sogenannte direkte Verkettung. Die Elemente dieser Liste können entweder in der Haupttabelle liegen oder auch nicht; im zweiten Fall heisst der ihnen zugewiesene Speicher gewöhnlich Ueberlaufbereich. Diese Methode ist sehr effektiv, obwohl sie den Nachteil hat, dass sekundäre Listen zu führen sind und dass jede Eintragung Platz für einen Zeiger (oder Index) zu ihrer Liste der kollidierenden Elemente zur Verfügung stellen muss.

Eine zweite Lösung zur Beseitigung von Kollisionen sieht völlig von Verknüpfungen ab und sucht statt dessen einfach unter anderen Eintragungen in der gleichen Tabelle, bis das Element oder ein freier Platz gefunden ist, in welchem Fall man annehmen kann, dass der angegebene Schlüssel in der Tabelle nicht vorhanden ist. Diese Methode heisst offene Adressierung [24]. Natürlich muss die Sequenz der Indizes bei weiteren Sondierungen für einen gegebenen Schlüssel immer die gleiche sein. Der Algorithmus für das Suchen in einer Tabelle kann dann folgendermassen skizziert werden:

```
h := H(k); i := 0;                                              (4.89)
repeat
   if T[h].key = k then "Element gefunden" else
   if T[h].key = "frei" then "Element nicht in Tabelle" else
   begin  {Kollision}
      i := i+1; h := H(k)+G(i)
   end
until "gefunden oder nicht in Tabelle (oder Tabelle voll)"
```

In der Literatur wurden mehrere Funktionen zur Auflösung von Kollisionen vorgeschlagen. Eine Uebersicht von Morris [23] über dieses Thema gab in einem beträchtlichen Mass Anlass zu weiteren Arbeiten auf diesem Gebiet. Die einfachste Methode ist, den nächsten Platz auszuprobieren - wobei man die Tabelle als zirkulär ansieht - bis entweder das Element mit dem angegebenen Schlüssel gefunden ist oder man auf einen leeren Platz stösst. Somit ist G(i) = i, und die zum Testen in diesem Fall verwendeten Indizes h[i] sind

```
h[0] = H(k)                                                     (4.90
h[i] = (h[0]+i) mod N , i = 1...N-1
```

Diese Methode heisst auch lineares Sondieren und hat den Nachteil, dass Eintragungen sich um primäre Schlüssel (Schlüssel, die beim Einfügen nicht kollidierten) ballen. Idealerweise sollte eine Funktion G gewählt werden, die die Schlüssel wiederum gleichförmig über die restliche Menge von Plätzen verteilt. In der Praxis erweist sich dies jedoch als zu teuer, und es werden Methoden vorgezogen, die einen Kompromiss bilden, indem sie einfach zu berechnen und immer noch besser als die lineare Funktion (4.90) sind. Eine davon verwendet eine quadratische Funktion mit folgender Sequenz der beim Sondieren verwendeten Indizes

```
h[0] = H(k)                                                     (4.91)
h[i] = (h[0]+i**2) mod N , (i > 0)
```

Bei Verwendung folgender Rekursions-Relationen (4.92) für h[i] = i**2 und d[i] = 2i+1 ist sogar das Quadrieren zur Berechnung des nächsten Index nicht nötig.

$$h[i+1] = h[i] + d[i] \qquad (i > 0) \qquad (4.92)$$
$$d[i+1] = d[i] + 2$$

mit h[0] = 0 und d[0] = 1 . Dies heisst quadratisches Sondieren und vermeidet im wesentlichen primäre Ballungen, obwohl praktisch keine zusätzlichen Berechnungen erforderlich sind. Ein kleiner Nachteil ist, dass dabei nicht alle Eintragungen der Tabelle untersucht werden, d.h. beim Einfügen kann es vorkommen, dass man keinen freien Platz findet, obwohl es noch welchen gibt. Tatsächlich wird bei quadratischem Sondieren mindestens die halbe Tabelle besucht, wenn ihre Grösse N eine Primzahl ist. Diese Aussage kann aus folgender Ueberlegung hergeleitet werden. Falls die i-te und j-te Sondierung auf die gleiche Eintragung der Tabelle fallen, so können wir dies ausdrücken durch die Gleichung

$$i^2 \bmod N = j^2 \bmod N \quad \text{oder}$$
$$(i^2 - j^2) = 0 \quad (\text{modulo } N)$$

Aufteilen der Differenz in zwei Faktoren ergibt

$$(i+j)(i-j) = 0 \quad (\text{modulo } N)$$

Da i <> j, muss entweder i oder j grösser N/2 sein, damit i+j = c*N mit positivem c ergibt.

Praktisch hat der erwähnte Nachteil keine Bedeutung, da N/2 Sondierungen und Kollisionsvermeidungen äusserst selten sind und nur dann vorkommen, wenn die Tabelle schon nahezu voll ist.

Als eine Anwendung der Speicherung durch Streuung (scatter storage technique), wird das Cross-Reference-Programm 4.5 neu geschrieben in Form von Programm 4.8. Die wesentlichen Unterschiede liegen in der Prozedur search und im Ersetzen des Zeigertyps wordref durch die Tabelle T vom Grundtyp word. Die hash-Funktion H ist der Modulus der Tabellengrösse; zur Behandlung der Kollision wurde quadratisches Sondieren gewählt. Für eine gute Leistung ist es wesentlich, dass die Tabellengrösse eine Primzahl ist.

Obwohl die Methode der Schlüssel-Transformation in diesem Fall sehr wirkungsvoll ist - wirkungsvoller als die Baumstrukturen - hat sie auch Nachteile. Nachdem man in Programm 4.8 den Text durchsucht und die Worte gesammelt hat, wollen wir diese Worte in alphabetischer Reihenfolge listen. Dies ist bei Verwendung einer Baumstruktur sehr

einfach, da sie eigentlich auf dem geordneten Suchbaum beruht. Bei Schlüssel-Transformationen ist dies aber nicht der Fall. Die volle Bedeutung des Wortes hashing kommt hier zum Ausdruck. Dem Ausdrucken der Tabelle muss nicht nur ein Sortierprozess vorangehen (der Einfachheit halber verwendet Programm 4.8 Sortieren mit direkter Auswahl), sondern es erweist sich auch als vorteilhaft, eingefügte Schlüssel durch Verkettung in einer speziellen Liste festzuhalten. Somit wird die bessere Leistung der hash-Methode, die nur den Prozess des Wiederfindens betrifft, zum Teil wieder durch zusätzliche Operationen aufgehoben, die zur vollständigen Erstellung eines geordneten Cross-Reference-Index erforderlich sind.

```
program crossref(f,output);
{erstellen einer Cross-Reference-Tabelle mittels einer
 Streutabelle}
label 13;
const c1 = 10;    {Wortlänge}
      c2 = 8;     {Zahlen pro Zeile}
      c3 =  6;    {Ziffern pro Zahl}
      c4 = 9999;  {max. Zeilennummer}
      p = 997;    {Primzahl}
      free = '          ';
type  index = 0..p;
      itemref = ↑item;
      word = record key: alfa;
                    first, last: itemref;
                    fol: index
             end ;
      item = packed record
                    lno: 0..9999;
                    next: itemref
             end ;
var i, top: index;
      k,k1: integer;
      n: integer;          {momentane Zeile}
      id: alfa;
      f: text;
      a: array [1..c1] of char;
      t: array [0..p] of word;       {Streutabelle}
```

```
procedure search;
   var h,d,i: index;
      x: itemref; f: boolean;
   {global Variable: t, id, top}
begin h := ord(id) div 4096 mod p;
   f := false; d := 1;
   new(x); x↑.lno := n; x↑.next := nil;
   repeat
      if t[h].key = id then
         begin {gefunden} f := true;
            t[h].last↑.next := x; t[h].last := x
         end else
      if t[h].key = free then
         begin {neue Eintragung} f := true;
            with t[h] do
            begin key := id; first := x; last := x; fol := top
            end ;
            top := h
         end else
         begin {Kollision} h := h+d; d := d+2;
            if h >= p then h := h-p;
            if d = p then
               begin writeln(' table overflow'); goto 13
               end
         end
   until f
end {search} ;

procedure printtable;
   var i,j,m: index;

   procedure printword(w: word);
      var l: integer; x: itemref;
   begin write(' ', w.key);
      x := w.first; l := 0;
      repeat if l = c2 then
            begin writeln;
               l := 0; write(' ':c1+1)
            end ;
         l := l+1; write(x↑.lno:c3); x := x↑.next
```

```
        until x = nil;
        writeln
    end {printword} ;

begin i := top;
    while i <> p do
    begin {suche kleinsten Schlüssel in verketteter Liste}
        m := i; j := t[i].fol;
        while j <> p do
            begin if t[j].key < t[m].key then m := j;
                j := t[j].fol
            end ;
        printword(t[m]);
        if m <> i then
            begin t[m].key := t[i].key;
                t[m].first := t[i].first; t[m].last := t[i].last
            end ;
        i := t[i].fol
    end
end {printtable} ;

begin n := 0; k1 := c1; top := p; reset(f);
    for i := 0 to p do t[i].key := free;
    while not eof(f) do
    begin if n = c4 then n := 0;
        n := n+1; write(n:c3);       {nächste Zeile}
        write(' ');
        while not eoln(f) do
        begin {durchsuche nichtleere Zeile}
            if f↑ in ['A'..'Z'] then
            begin k := 0;
                repeat if k < c1 then
                    begin k := k+1;  a[k] := f↑;
                    end ;
                    write(f↑); get(f)
                until not (f↑ in ['A'..'Z','0'..'9']);
                if k >= k1 then k1 := k else
                repeat a[k1] := ' ';  k1 := k1-1
                until k1 = k;
                pack(a,1,id); search;
```

```
            end else
            begin {teste auf Anführungszeichen und Kommentar}
               if f↑ = '''' then
                  repeat write(f↑); get(f)
                  until f↑ = '''' else
               if f↑ = '{' then
                  repeat write(f↑); get(f)
                  until f↑ = '}' :
               write(f↑); get(f)
            end
         end ;
         writeln; get(f)
      end ;
   13: page; printtable
end .
```

Programm 4.8

4.6.3. Analyse der Schlüssel-Transformation

Einfügen und Wiederauffinden durch Schlüssel-Transformation erbringt offenbar eine schlechte Leistung im schlimmsten Fall. Es ist gut möglich, dass ein Suchargument bei den Sondierungen genau auf alle besetzten Plätze stösst und somit dauernd den gewünschten (oder freien) Platz verpasst. Eigentlich benötigt jeder, der die hash-Technik verwendet, ein beträchtliches Vertrauen in die Gesetze der Wahrscheinlichkeitstheorie. Wir wollen sichergehen, dass im Mittel die Zahl der Sondierungen klein ist. Die folgende wahrscheinlichkeitstheoretische Ueberlegung zeigt, dass sie sogar sehr klein ist.

Wir wollen einmal mehr annehmen, dass alle möglichen Schlüssel gleich wahrscheinlich sind und dass die hash-Funktion H sie gleichförmig auf den Bereich der Tabellenindizes verteilt. Es sei dann ein Schlüssel in eine Tabelle der Grösse n einzufügen, die bereits k Elemente enthält. Die Wahrscheinlichkeit, auf den ersten Hieb einen freien Platz zu finden, ist dann 1-k/n. Dies ist auch die Wahrscheinlichkeit p[1], dass nur ein einziger Vergleich benötigt wird. Die Wahrscheinlichkeit genau einer zweiten Sondierung

ist gleich der Wahrscheinlichkeit einer Kollision beim ersten Versuch multipliziert mit der Wahrscheinlichkeit, das nächste Mal einen freien Platz zu finden. Allgemein erhalten wir die in (4.93) dargestellte Wahrscheinlichkeit p[i] für ein Einfügen mit i Sondierungen.

$$
\begin{aligned}
p[1] &= (n-k)/n \\
p[2] &= k/n * (n-k)/(n-1) \\
p[3] &= k/n * (k-1)/(n-1) * (n-k)/(n-2) \\
&\ldots \\
p[i] &= k/n * (k-1)/(n-1) * (k-2)/(n-2) * \ldots \\
&\quad * (k-i+2)/(n-i+2) * (n-k)/(n-i+1)
\end{aligned}
\tag{4.93}
$$

Die zu erwartende Zahl von Sondierungen, die beim Einfügen des (k+1)-ten Schlüssels benötigt wird, beträgt daher

$$
\begin{aligned}
E[k+1] &= 1*p[1] + 2*p[2] + \ldots + (k+1)*p[k+1] \\
&= 1*(n-k)/n + 2*(k/n)*(n-k)/(n-1) + \ldots \\
&\quad + (k+1) * (k/n * (k-1)/(n-1) * (k-2)/(n-2) * \ldots \\
&\quad * 1/(n-k+1)) \\
&= (n+1)/(n-k+1)
\end{aligned}
\tag{4.94}
$$

Da die zum Einfügen eines Elementes benötigte Zahl von Sondierungen identisch ist mit der zum Finden notwendigen Zahl von Sondierungen, kann das Ergebnis (4.94) zur Berechnung der mittleren Zahl E der zum Zugriff eines beliebigen Schlüssels in einer Tabelle benötigten Sondierungen verwendet werden. Die Tabellengrösse sei wiederum mit n bezeichnet, und m sei die Zahl der momentan in der Tabelle vorhandenen Schlüssel. Dann ist

$$
\begin{aligned}
E &= (1/m) * \sum_{i=1}^{m} E[k] = (n+1)/m * \sum_{k=1}^{m} 1/(n-k+2) \\
&= (n+1)/m * (H[n+1] - H[n-m+1])
\end{aligned}
\tag{4.95}
$$

wobei

$$H[n] = 1 + 1/2 + \ldots + 1/n$$

die harmonische Funktion ist. H[n] kann approximiert werden durch $H[n] = \ln(n)+g$ mit der Eulerschen Konstanten g. Setzen wir ausserdem $a = m/(n+1)$, so erhalten wir

$$E = (1/a) * (\ln(n+1) - \ln(n-m+1)) \qquad (4.96)$$
$$= (1/a) * \ln((n+1)/(n+1-m)) = (-1/a) * \ln(1-a)$$

a ist ungefähr der Quotient der belegten und verfügbaren Plätze, der sogenannte Auslastungsfaktor; a = 0 bedeutet eine leere Tabelle, a = n/(n+1) eine volle Tabelle. Die zu erwartende Zahl E von Sondierungen zum Wiederfinden oder Einfügen eines beliebig gewählten Schlüssels ist in Tabelle 4.6 als Funktion des Auslastungsfaktors a aufgeführt.

a	E
0.1	1.05
0.25	1.15
0.5	1.39
0.75	1.85
0.9	2.56
0.95	3.15
0.99	4.66

Tabelle 4.6

Die numerischen Werte sind wirklich überraschend und erklären die aussergewöhnlich gute Leistung der Methode der Schlüssel-Transformation. Selbst bei zu 90% gefüllter Tabelle sind im Mittel nur 2.56 Sondierungen zum entweder Lokalisieren eines Schlüssels oder Finden eines leeren Platzes notwendig! Dieser Wert hängt insbesondere nicht von der absolut vorhandenen Zahl von Schlüsseln ab, sondern ausschliesslich vom Auslastungsfaktor.

Obige Analyse beruht auf einer Methode zur Behandlung von Kollisionen, die die Schlüssel gleichförmig über die restlichen Plätze verstreut. In der Praxis verwendete Methoden ergeben eine leicht schlechtere Leistung. Ausführliche Analyse für <u>lineares Sondieren</u> ergibt eine in (4.97) angegebene Zahl zu erwartender Sondierungen [25].

$$E = (1 - a/2)/(1-a) \qquad (4.97)$$

In Tabelle 4.7 sind einige numerische Werte für E(a) aufgeführt.

a	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Tabelle 4.7

Die selbst mit der einfachsten Methode zur Behandlung der Kollision erhaltenen Werte sind so gut, dass die Versuchung gross ist, die Schlüssel-Transformation (hashing) als Allheilmittel zu betrachten. Dies ist besonders der Fall, da ihre Leistung sogar besser ist als die der ausgeklügeltsten Baumorganisation, mindestens in bezug auf die zum Wiederauffinden und Einfügen notwendigen Schritte. Es ist daher wichtig, einige Nachteile der hash-Methode ausdrücklich zu erwähnen, selbst wenn sie bei unvoreingenommener Betrachtung selbstverständlich sind.

Der sicherlich grösste Nachteil gegenüber Techniken mit dynamischer Speicherzuweisung ist die feste Grösse der Tabelle, die dem momentanen Bedarf nicht angepasst werden kann. Eine ziemlich gute a priori Abschätzung der Zahl der zu ordnenden Datenelemente ist daher Voraussetzung, wenn entweder schlechte Speicherausnutzung oder schlechte Leistung (oder sogar Ueberlauf) vermieden werden soll. Sogar wenn die genaue Zahl der Elemente bekannt ist - ein sehr seltener Fall - muss für eine gute Leistung die Dimension der Tabelle etwas (ungefähr 10%) zu gross gewählt werden.

Die zweitgrösste Schwäche der Speicherung durch Streuung tritt zutage, wenn Schlüssel nicht nur einzufügen und wiederzufinden, sondern auch zu löschen sind. Denn Löschen von Eintragungen in einer hash-Tabelle ist äusserst umständlich, wenn nicht direkte Verkettung in einem separaten Ueberlaufbereich verwendet wird. Es ist daher fair, Organisationen mit Bäumen immer noch attraktiv zu nennen und auch vorzuziehen, wenn der Umfang der Daten weitgehend unbekannt, stark veränderlich und zeitweise sogar abnehmend ist.

Uebungen

4.1. Wir wollen den Begriff eines rekursiven Typs

```
rectype T = T0
```

zur Bezeichnung der Vereinigung der durch den Typ T0 definierten Werte und den einen Wert none einführen, d.h.

```
T = T0 union {none}
```

Die Definition (4.3) des Typs Stammbaum z.B. vereinfacht sich dann zu

```
rectype Stammbaum = record name: alfa;
                           father,mother: Stammbaum
                    end
```

Welches Muster für die Speicherung der rekursiven Struktur entspricht Fig. 4.2?

Wahrscheinlich würde die Implementation eines solchen Hilfsmittels dynamische Speicherzuweisung verwenden, und die Felder father und mother in obigem Beispiel würden automatisch erzeugte und vom Programmierer versteckte Zeiger enthalten. Welche Schwierigkeiten stellen sich bei dieser Realisierung?

4.2 Man definiere die im letzten Paragraphen von Abschnitt 4.2 beschriebene Datenstruktur mit Records und Zeigern. Kann man diese Familienverhältnisse auch mit den in Uebung 4.1 vorgeschlagenen rekursiven Typen darstellen?

4.3. Eine Warteschlange Q (first-in-first-out) sei mit Elementen vom Typ T0 als verkettete Liste implementiert. Man definiere eine geeignete Datenstruktur, Prozeduren zum Einfügen und Entnehmen eines Elementes aus Q und eine Funktion zum Testen, ob die Schlange leer ist. Diese Prozeduren sollten einen eigenen Mechanismus für wirtschaftliche Wiederverwendung des Speichers enthalten.

4.4. Man nehme an, dass die Records einer verketteten Liste ein Schlüsselfeld vom Typ integer enthalten. Es sei ein Programm zu schreiben, das die Liste nach aufsteigenden Werten der Schlüssel sortiert. Weiter ist eine Prozedur zur Umkehrung der Liste zu erstellen.

4.5. Zirkuläre Listen (Fig. 4.54) werden normalerweise mit einem sogenannten Listenkopf aufgebaut

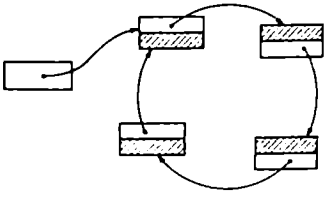


Fig. 4.54: Zirkuläre Liste mit Listenkopf

Aus welchem Grund wird ein solcher Kopf eingeführt? Man schreibe Prozeduren zum Einfügen, Löschen und Suchen eines durch einen gegebenen Schlüssel bezeichneten Elementes. Diese sind einmal unter der Annahme der Existenz eines Kopfes zu schreiben und einmal ohne.

4.6. Eine Zweiwegliste ist eine Liste von Elementen, die in beiden Richtungen verknüpft sind.

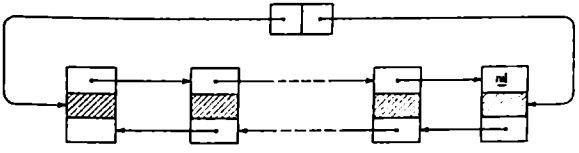


Fig. 4.55: Zweiwegliste mit Listenkopf

Beide Ketten gehen von einem Kopf aus. Entsprechend zu Aufgabe 4.5 ist ein Satz von Prozeduren zum Suchen, Einfügen und Löschen von Elementen zu schreiben.

4.7. Arbeitet Programm 4.2 korrekt, wenn ein gewisses Paar <x,y> mehr als einmal in der Eingabe vorkommt?

4.8. Die Meldung "diese Menge ist nicht teilweise geordnet" von Programm 4.2 ist in vielen Fällen nicht sehr hilfreich. Man erweitere das Programm, so dass es eine Sequenz von Elementen ausdruckt, die eine Schleife bilden, falls eine existiert.

4.9. Man schreibe ein Programm, das einen Programmtext liest, alle Definitionen und Aufrufe von Prozeduren (Subroetinen) feststellt und versucht, zwischen diesen Subroutinen eine topologische Ordnung aufzustellen. Es gelte P < Q, wenn P von Q aufgerufen wird.

4.10. Man zeichne den von Programm 4.3 erstellten Baum, wenn die Eingabe aus den n+1 Zahlen

n,1,2,3,...,n

besteht.

4.11. In welcher Reihenfolge findet man die Knoten beim Durchlaufen des Baumes von Fig. 4.23 in preorder, inorder und postorder?

4.12. Man finde das Bildungsgesetz für die Folge der n Zahlen, die unter Verwendung von Programm 4.4 einen völlig ausgeglichenen Baum ergeben.

4.13. Man betrachte die folgenden beiden Anweisungen für das Durchlaufen eines binären Baumes.

(a) 1) Durchlaufe den rechten Teilbaum.
2) Besuche die Wurzel.
3) Durchlaufe den linken Teilbaum.

(b) 1) Besuche die Wurzel.
2) Durchlaufe den rechten Teilbaum.
3) Durchlaufe den linken Teilbaum.

Gibt es eine einfache Beziehung zwischen den sich bei diesen Anweisungen ergebenden Reihenfolge des Knotendurchlaufs und den durch die drei im Text definierten Anweisungen entstehenden Reihenfolgen?

4.14. Man definiere eine Datenstruktur zur Darstellung eines n-ären Baumes und schreibe eine Prozedur, die diesen n-ären Baum durchläuft und einen binären Baum mit den gleichen Elementen erzeugt. Man nehme an, dass der in jedem Element gespeicherte Schlüssel k Worte und jeder Zeiger ein Speicherwort belegt. Wie gross ist der Speichergewinn bei Verwendung eines binären Baumes gegenüber einem n-ären Baum?

4.15. Man nehme an, dass ein Baum entsprechend folgender Definition einer rekursiven Datenstruktur aufgebaut ist (siehe Uebung 4.1).

```
reqtype tree = record x: integer;
                      left,right: tree
               and
```

Man formuliere eine Prozedur zum Finden eines Elementes mit gegebenem Schlüssel x, die eine Operation P auf diesem Element ausführt.

4.16. In einem File-System ist ein Katalog aller Files in Form eines geordneten binären Baumes organisiert. Jeder Knoten bezeichnet ein File und enthält den File-Namen sowie unter anderem das Datum des letzten Zugriffs als ganze Zahl verschlüsselt.

Man schreibe ein Programm, das diesen Baum durchsucht und alle Files löscht, zu denen seit einem bestimmten Datum nicht mehr zugegriffen wurde.

4.17. In einer Baumstruktur wird die Häufigkeit des Zugriffs zu jedem Element empirisch gemessen, indem man jedem Knoten einen

Zugriffszähler zuteilt. In gewissen Zeitabständen wird die Baumorganisation angepasst, indem man den Baum durchläuft und unter Verwendung von Programm 4.4 einen neuen Baum erzeugt, wobei man die Schlüssel nach abnehmendem Wert des Zugriffszählers ordnet. Man schreibe ein Programm, das diese Reorganisation durchführt. Ist die mittlere Weglänge in diesem Baum gleich, schlechter oder sehr viel schlechter als die in einem optimalen Baum?

4.18. Die in Abschnitt 4.4.5 beschriebene Methode zur Analyse des Algorithmus des Einfügens in einen Baum kann auch zur Berechnung der zu erwartenden Zahlen C von Vergleichen und M von Bewegungen (Austauschen) verwendet werden, die beim Quicksort (Programm 2.10) zum Sortieren der n Elemente eines Array ausgeführt werden unter der Annahme, dass alle n! Permutationen der n Schlüssel {1,2,...,n} gleich wahrscheinlich sind. Man finde die Analogie und bestimme C[n] und M[n].

4.19. Man zeichne den ausgeglichenen Baum mit 12 Knoten, der die maximale Höhe aller Bäume mit 12 Knoten hat. In welcher Reihenfolge müssen die Knoten eingegeben werden, damit Prozedur (4.63) diesen Baum erzeugt?

4.20. Man finde eine Eingabefolge von Schlüsseln, so dass Prozedur (4.63) jeden der vier Ausgleichsvorgänge (LL, LR, RR, RL) mindestens einmal ausführt. Wie gross ist die minimale Länge für eine solche Folge?

4.21. Man finde einen ausgeglichenen Baum mit Schlüsseln 1...n und eine Permutation dieser Schlüssel, so dass Anwendung der Prozedur (4.64) zum Löschen jede der vier Ausgleichsroutinen einmal ausführt. Welche Sequenz hat die minimale Länge n?

4.22. Welches ist die mittlere Weglänge des Fibonacci-Baumes T[n]?

4.23. Man schreibe ein Programm, das entsprechend dem Algorithmus von Hu-Tucker durch Auswahl eines Zentroid als Wurzel den nahezu optimalen Baum generiert.

4.24. Man nehme an, dass die Schlüssel 1,2,3,... in einen leeren B-Baum der Ordnung 2 eingefügt werden (Programm 4.7). Welche Schlüssel verursachen Aufteilungen von Seiten? Welche Schlüssel verursachen ein Wachsen der Höhe des Baumes?

Welche Schlüssel veranlassen beim Löschen der Schlüssel in der gleichen Reihenfolge eine Zusammenlegung (und Freigabe) von Seiten, welche Schlüssel verursachen ein Schrumpfen der Höhe? Man beantworte die Frage für

(a) ein Schema zum Löschen mit Ausgleichen (wie in Programm 4.7) und

(b) ein Schema ohne Ausgleichen (bei Unterlauf wird ein einzelnes Element aus einer benachbarten Seite geholt).

4.25. Man schreibe ein Programm zum Suchen, Einfügen und Löschen von Schlüsseln in einem binären B-Baum. Man verwende die Knoten-Typdefinition (4.84). Das Schema zum Einfügen ist in Fig. 4.51 gezeigt.

4.26. Man suche eine Sequenz von Eingabe-Schlüsseln, die, ausgehend vom leeren symmetrischen binären B-Baum, Prozedur (4.87) veranlasst, alle vier Ausgleichsvorgänge (LL, LR, RR, RL) mindestens einmal auszuführen. Welches ist die kürzeste solche Sequenz?

4.27. Man schreibe eine Prozedur zum Löschen von Elementen in einem symmetrischen binären B-Baum. Dazu ist ein Baum und eine kurze Sequenz zu löschender Schlüssel zu suchen, für die alle vier Ausgleichssituationen mindestens einmal auftreten.

4.28. Man vergleiche die Leistungen des Algorithmus zum Einfügen und Löschen bei binären Bäumen, bei AVL-ausgeglichenen Bäumen und symmetrischen binären B-Bäumen auf der eigenen Rechenanlage. Man untersuche besonders die Auswirkungen des Packens von Daten, d.h. die Wahl einer wirtschaftlichen Datendarstellung mit nur 2 Bits für die Ausgleichsinformation in jedem Knoten.

4.29. Man ändere den Ausgabe-Algorithmus von Programm 4.6 derart, dass er zur Darstellung eines symmetrischen binären B-Baumes mit horizontalen und vertikalen Kanten verwendet werden kann.

4.30. Wenn die mit jedem Schlüssel verknüpfte Menge an Information relativ gross ist (im Vergleich zum Schlüssel selbst), sollte diese Information nicht in einer hash-Tabelle gespeichert werden. Man erkläre warum und schlage ein Schema zur Darstellung einer solchen Datenmenge vor.

4.31. Man betrachte den Vorschlag zur Lösung des Ballungsproblems durch Konstruktion eines Ueberlaufbaumes statt einer Ueberlaufliste, d.h. einer Organisation kollidierender Schlüssel als Baumstrukturen. Dabei kann jede Eintragung in einer Streutabelle (hash-Tabelle) als Wurzel eines (möglicherweise leeren) Baumes aufgefasst werden (tree-hashing).

4.32. Man entwerfe ein Schema zur Ausführung von Eintragungen und Streichungen in einer Streutabelle unter Verwendung quadratischer Schrittweiten zur Auflösung von Kollisionen. Man vergleiche dieses Schema experimentell mit der Organisation eines binären Baumes, indem man Zufallsfolgen von Schlüsseln zum Einfügen und Löschen verwendet.

4.33. Der primäre Nachteil der Technik der hash-Tabelle ist der, dass die Grösse der Tabelle zu einer Zeit festzulegen ist, zu der die tatsächliche Zahl von Eintragungen nicht bekannt ist. Nehmen Sie an, dass auf Ihrer Rechenanlage ein Mechanismus zur dynamischen

Speicherverwaltung zur Verfügung steht, der erlaubt, jederzeit Speicher zu verlangen. Dann wird, wenn eine hash-Tabelle H voll (oder nahezu voll) ist, eine grössere Tabelle H' generiert, und alle Schlüssel in H werden transferiert nach H', wonach der Speicher für H dem Mechanismus wieder zur Verfügung gestellt werden kann. Diese Methode heisst rehashing.

Man schreibe ein Programm, das eine erneute Streuung (rehash) einer Tabelle H der Grösse n in eine Tabelle H' der Grösse n' durchführt.

4.34. Sehr oft sind Schlüssel keine ganzen Zahlen sondern Buchstabenfolgen. Diese Worte können in ihrer Länge sehr stark variieren und sind daher nicht leicht und wirtschaftlich in Schlüsselfeldern fester Grösse zu speichern. Man schreibe ein Programm, das mit einer hash-Tabelle und variabler Schlüssellänge arbeitet.

Anhang A. Der ASCII-Zeichensatz

b6	0	0	0	0	1	1	1	1
b5	0	0	1	1	0	0	1	1
b4	0	1	0	1	0	1	0	1
b3-b0								
0000	nul	dle		0	@	P	`	p
0001	soh	dc1	!	1	A	Q	a	q
0010	stx	dc2	"	2	B	R	b	r
0011	etx	dc3	#	3	C	S	c	s
0100	eot	dc4	$	4	D	T	d	t
0101	enq	nak	%	5	E	U	e	u
0110	ack	syn	&	6	F	V	f	v
0111	bel	etb	'	7	G	W	g	w
1000	bs	can	(	8	H	X	h	x
1001	ht	em	)	9	I	Y	i	y
1010	lf	sub	*	:	J	Z	j	z
1011	vt	esc	+	;	K	[	k	{
1100	ff	fs	,	<	L	\	l	|
1101	cr	gs	-	=	M	]	m	}
1110	so	rs	.	>	N	^	n	~
1111	si	us	/	?	O	_	o	del

Steuer-zeichen — Schriftzeichen

eingeschränkter ASCII-Zeichensatz

Bedeutung der Steuerzeichen

Layout characters
- bs backspace
- ht horizontal tabulation
- lf line feed
- vt vertical tabulation
- ff form feed
- cr carriage return

Ignore characters
- nul null character
- can cancel
- sub substitute
- del delete

Separator characters
- fs file separator
- gs group separator
- rs record separator
- us unit separator

Escape characters
- so shift-out
- si shift-in
- esc escape

Medium control characters
- bel ring bell
- dc1-dc4 device control
- em end of medium

Communication control characters
- soh start of heading
- stx start of text
- etx end of text
- eot end of transmission
- enq enquiry
- ack acknowledgement
- nak negative acknowledgement
- dle data link escape
- syn synchronous idle
- etb end of transmission block

Anhang B, Verzeichnis der Programme

Array, Durchsuchen (1.15, 1.16), 32, 33
Baum, Durchlaufen (4.43)-(4.45)
Durchsuchen (4.46)-(4.47)
Durchsuchen und Einfügen (Pr 4.4)
-, nicht rekursiv (4.51)
Löschen (4.52)
-, ausgeglichener
Durchsuchen und Einfügen (4.63)
Löschen (4.64)
-, vollständig ausgeglichener (Pr 4.3), 265-266
-, optimaler Suchbaum
Erstellen und Darstellen (Pr 4.6)
B-Baum
Durchsuchen,Einfügen und Löschen (Pr 4.7)
-, symmetrischer
Durchsuchen und Einfügen (4.87)
binäres Suchen (1.17), 33
Cross-Reference-Liste (Pr 4.5)
-, mit Streu-Tabelle (Pr 4.8)
File, Aendern (1.57)-(1.61), 80-83
Lesen (1.53), 67
Schreiben (1.52), 66
-, Text lesen und schreiben (1.54), (1.55), 72
Hilbert-Kurven (Pr 3.1) 186-187
Liste, direktes Durchsuchen und Einfügen
(Pr 4.1) (4.21)-(4.26), 239-247
Durchlaufen (4.17), 237
Generieren (4.13), 234
Optimale Auswahl (Pr 3.7), 216-218
Potenzen von 2 (Pr 1.1), 35
Problem der acht Damen (Pr 3.4, Pr 3.5), 201, 203-204
Readreal (Pr 1.3), 74-77
Scanner (Pr 1.2), 48-49
Sierpinski-Kurven (Pr 3.2), 189-190
Sortieren, Bubblesort (Pr 2.4), 100
Auswählen, direktes (Pr 2.3), 97-98
Einfügen, binäres (Pr 2.2), 95
-, direktes (Pr 2.1), 94
Heapsort (Pr 2.8), 112
Mehrphasen-Sortieren (Pr 2.16), 162-165
Mischsortieren, ausgeglichenes (Pr 2.15), 149-151
-, direktes (Pr 2.13), 133-134
-, natürliches (Pr 2.14), 141-143
Partition (Pr 2.9), 114
Quicksort, rekersiv (Pr 2.10), 117
-, nicht rekursiv (Pr 2.11), 118
Shakersort (Pr 2.5) 101-102
Shellsort (Pr 2.6), 105-106
Sift (Pr 2.7), 110
topologisches Sortieren (Pr 4.2), 255-256
Verteilung der Läufe (Pr 2.17), 169-170
Stabile Heirat (Pr 3.6), 209-211
Stundenplan (1.28)-(1.30), 50-51
Weg des Springers (Pr 3.3), 196-197
Writereal (Pr 1.4), 77-79

Literatur

[1] Adelson-Velskii , G.M.; Landis , E.M.: Ein Algorithmus zur Informationsorganisation (russisch). Doklady Akademiia Nauk SSSR 146 (1962) 263-266; English translation in Soviet Maths. 3, 1259-1263

[2] Bayer , R.: Binary B-trees for Virtual Memory. Proc. 1971 ACM SIGFIDED Workshop, San Diego (Nov. 1971) 219-235

[3] Bayer , R; McCreight , E.: Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1, No. 3 (1972) 173-189

[4] Bayer , R.: Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. Acta Informatica 1, No. 4 (1972) 290-306

[5] Betz , B.K.; Carter , W.C.: ACM National Conf. 14 (1959) Paper 14

[6] Dijkstra , E.W.: Notes on Structured Programming. In: Structured Programming. Academic Press, London, New York (1972)

[7] Floyd , R.W.: Treesort. (Algorithms 113 and 243) Comm. ACM 5, No. 8 (1962) 434 und Comm. ACM 7, No. 12 (1964) 701

[8] Gilstad , R.L.: Polyphase Merge Sorting - An Advanced Technique. Proc. AFIPS Eastern Jt. Comp. Conf. 18 (1960) 143-148

[9] Hoare , C.A.R.: Quicksort. Comp. J. 5, No. 1 (1962) 10-15

[10] Hoare , C.A.R.: An Axiomatic Basis for Computer Programming. Comm. ACM 12, No. 10 (1969) 35-63

[11] Hoare , C.A.R.: Proof of a Program: FIND. Comm. ACM 13, No. 1 (1970) 39-45

[12] Hoare , C.A.R.: Proof of a Recursive Program: QUICKSORT. Comp. J. 14, No. 4 (1971) 391-395

[13] Hoare , C.A.R.: Notes on Data Structuring. In: Structured Programming. Dahl, O.J.; Dijkstra, E.W.; Hoare, C.A.R.; Academic Press, London, New York (1972) 83-174

[14] Hu , T.C.; Tucker , A.C.. SIAM J. Applied Math. 21, No. 4 (1971) 514-532

[15] Jensen , K; Wirth , N.: PASCAL, User Manual and Report. Lecture Notes in Computer Science, Vol. 18. Springer-Verlag, Berlin, Heidelberg und New York (1974)

[16] Knuth , D.E.: Optimum Binary Search Trees. Acta Informatica 1, No. 1 (1971) 14-25

[17] Knuth , D.E.: The Art of Computer Programming. Bd. 1, 95-99

[18] Knuth , D.E.: The Art of Computer Programming. Bd. 3. Reading, Mass.: Addison-Wesley (1973)

[19] Lorin , H.: A Guided Bibliography to Sorting. IBM Syst. J. 10, No. 3 (1971) 244-254

[20] Maurer W.D.: An Improved Hash Code for Scatter Storage. Comm. ACM 11, No. 1 (1968) 35-38

[21] McVitie , D.G.; Wilson , L.B.: The Stable Marriage Problem. Comm. ACM 14, N. 7 (1971) 486-492

[22] McVitie , D.G.; Wilson , L.B.: Stable Marriage Assignment for Unequal Sets. BIT 10 (1970) 295-309

[23] Morris , R.: Scatter Storage Techniques. Comm. ACM 11, No. 1 (1968) 38-43

[24] Peterson , W.W.: Addressing for Random-Access Storage. IBM J. Res. & Dev. 1 (1957) 130-146

[25] Schay , G; Soruth , W.: Analysis of a File Addressing Method. Comm. ACM 5, N. 8 (1962) 459-462

[26] Shell , D.L.: A Highspeed Sorting Procedure. Comm. ACM 2, No. 7 (1959) 30-32

[27] Singleton , R.C.: An Efficient Algorithm for Sorting with Minimal Storage. (Algorithm 347) Comm. ACM 12, No. 3 (1969) 185

[28] van Emden , M.H.: Increasing the Efficiency of Quicksort. (Algorithm 402) Comm. ACM 13, No. 9 (1970) 563-566, 693

[29] Walker , W.A.; Gotlieb , C.C.: A top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees. In: Graph Theory and Computing. Academic Press, New York (1972) 303-323

[30] Williams , J.W.J.: Heapsort. (Algorithm 232) Comm. ACM 7, No. 6 (1964) 347-348

[31] Wirth , N.: The Programming Language PASCAL. Acta Informatica 1, No. 1 (1971) 35-63

[32] Wirth , N.: Program Development by Stepwise Refinement. Comm. ACM 14, No. 4 (1971) 221-227

[33] Wirth , N.: Systematisches Programmieren. Teubner Studienbücher Informatik Bd. 17, Stuttgart (1972)

[34] / Space Filling Curves, or How to Waste Time on a Plotter. Software-Practice and Experience 1, No. 4 (1971) 403-440

[35] / Revolution in Programming. Datamation, (Dec 1973) 50-61

Sachverzeichnis

Abstraktion 15
Adelson-Velskii, G.M. 289
Adresse 53, 226
Aendern eines Files 79
alfa (Typ) 55
ALGOL 11, 18, 30
Array-Selektor 31
ASCII 27,71
Auffüllen 54
Aufzählung 21, 33
Ausdruck 223, 261, 262
Ausgeglichenheit (Baum) 289
Auslastungsfaktor 368
AVL-Baum 289, 348

Backtracking 197
Ballung 353
Baum 176, 257
-, Durchlaufen 198
-, Einfügen 272, 291, 329, 342, 345
-, Grad 259
-, Höhe 259
-, Löschen 283, 298, 332
-, Stufe 259
-, Suchen 272, 329, 340, 345
-, Tiefe 259
B-Baum 325, 326
BB-Baum 340
Bayer, R. 326, 340, 341
binärer Baum 261
binärer B-Baum 340
binäres Suchen 33, 86
Blatt 259
Boolean 26
branch and bound 218

case-Anweisung 44
char 27
charakteristische Funktion 58
chr(x) 28, 35
Coroutine 171
Cross-Reference-Liste 277

Datendarstellung 16, 53
Dijkstra, E.W. 7
direkter Zugriff (File) 69
Diskriminator-Feld 43
div 25
Durchlauf 129, 136
dynamische Speicherzuteilung 226
einelementige Menge 47
- Sequenz 61
Einphasen-Sortieren 129
eof(f) 65
eoln(f) 72
Euler, L. 5, 99, 288, 359

Fakultät 177, 188
Fibonacci-Baum 290
- -Zahlen 154, 181
File-Operator 64
- -Struktur 67, 68
first(x) 61
Floyd, R.W. 118

for-Anweisung 35
FORTRAN 30, 179

Gauss, C.F. 198
gemeinsame Daten 227
geordnete Liste 241
get(f) 65
gewichtete Weglänge 304
Gilstad, R.L. 151
Gotlieb, C.C. 310
Grad der Ausgeglichenheit 293
Grundtyp 21, 30, 45

Häufigkeitsindex 238
harmonische Zahlen 98, 287, 359
Hash-Funktion 351
hashing 351
Heap 188, 166
Heapsort 107, 112
Heckenbaum 345
Hilbert, D. 183
Hilbert-Kurve 183, 185
Hoare, C.A.R. 7, 8, 113, 120, 122
homogene Struktur 21, 30
Hu, T.C. 310

Indextyp 30
indiziertes File 68, 69
Infix-Notation 269
inorder 268
input 70
integer 25
ISO 27, 71

Kardinalität 21, 23, 32, 36, 46, 53, 59
Kartesisches Produkt 36
Kaskaden-Misch-Sortieren 175
Knuth, D.E. 91, 103, 106
Kollision 350
Konstruktor 22, 31, 37

Landis, E.M. 289
Latenzzeit 69
Lauf 135
lexikographischer Baum 277
Liste 233
-, Durchlaufen 236
-, Einfügen 235, 239
-, Generieren 234
-, Löschen 236
-, Suchen 236

Marke 33, 93, 238, 240
Matrix 34
McCreight, E. 340
McVitie, D.G. 212
Mehrphasen-Sortieren 151
Mengenoperationen 47
-, Durchschnitt 47
-, Differenz 47
-, Vereinigung 47
Mengenstruktur 45
Methoden der Strukturierung 21

Mischen 128
-, ausgeglichenes 129, 143
-, direktes 128
-, natürliches 135
Mischsortieren 128, 133, 144
mittleres Element 121
mod 25

Nachfolger 259
new(p) 228
nil 229

offene Adressierung 352
Offset 56
optimaler Baum 305
ord(x) 28, 35
output 78

pack 56
Packen 55, 57
PASCAL 8, 11, 18, 19, 28, 21, 74, 125
Phase 129, 136
Postfix-Notation 269
postorder 268
Potenzmenge 45
Prefix-Notation 269
preorder 268
Programmiersprache 11, 17
Programmschema 66, 72, 178, 198
Pseudo-Lauf 156
put(f) 64

Quicksort 113, 114

random-access 38
read 66
readln 72
real 26
Record-Diskriminator 43
- -Selektor 38
rehashing 369
Rekursion 176
- Schema 184
rekursiver Datentyp 223
reset(f) 64, 65
rest(x) 61
rewrite(f) 64

BBB-Baum 345
Scanner 47
Schlange (Queue) 233
Schleifeninvariante 33, 48
Schlüssel 98
- transformation 349, 358
Schrittweise Verfeinerung 79
Segment (File) 68
selbstorganisierende Liste 245
selektives Aendern 33, 38, 238
Selektor 23, 31, 38, 48, 42
Seite (Baum) 325
-, Aufteilung 328
-, Ausgleich 333
-, Zusammenlegen 333
Sequenz 61
Set 45
Shakersort 101, 102
Shell, D.L.. 103
Shellsort 105, 106
Sierpinski-Kurve 187
skalare Typen 21, 23
Sondieren, lineares 353, 360
-, quadratisches 354
Sortieren 88
- durch Austausch 99
- durch Auswahl 96
- durch binäres Einfügen 95
- durch Einfügen 92
-, externes 89
-, internes 89
-, stabiles 91
Sortiermethoden, direkte 125
-, logarithmische 125
Standardtypen 21, 25
Streuspeicherung 351
String 135
Suchbaum 270
symmetrisch binärer Baum 345

Teilbaum 257
teilweise Ordnung 248, 249
Termination 178
topologisches Sortieren 248
Tucker, A.C. 310
Typ (Daten-) 19
Typdefinition 19, 20, 24, 29, 30, 37, 43, 62, 227, 228

unpack 56
Ueberlaufbereich 352

variante Record-Struktur 43
varianter Record 42
Verkettung 61
-, direkte 352
Verteilung der Läufe 136, 156, 166
Vielweg-Baum 261, 323
vollständig ausgeglichen 265
Vorgänger 259

Walker, W.A. 310
Weglänge, innere 259
-, äussere 259,260
Williams, J. 100
Wilson, L.B. 212
Wort 53, 54
Wortgrösse 53
write 66
writeln 71
Wurzel 258

Zeiger (Typ) 226
Zentroid 310
Zerlegung 114, 120, 122

2-3-Baum

Korrekturen zu "Irth, Algorithmen und Datenstrukturen

Seite	Zeile	Korrektur
7	10	intellektuelle statt intellektülle
12	14	englischen statt Englischen
14	2	Rekursive statt Rekersive
20	5,6	ohne dass der Rechenprozess durchgeführt werden muss. statt ohne den ... zu müssen.
22	-10	viele statt alle
24	1	Soldat, Gefreiter statt Privat-mann, Obergefreiter
	4	Struktur statt structur
26	10	not p statt not q
29	14	Buchstabe statt char
	-6	ist, erst bei statt ist, bei
33	-6	> statt <
36	21	Zivilstand einer statt Zivilstandeiner
42	-2,	
	-10	r, phi: real statt r: real; phi: Winkel
46	-12	[manuell] statt manuell
55	9	ersetzt durch u = n*s
56	16	ersetzt durch wobei c-d <= d-e. Dann ist
67	-1	T(i) statt Ti
68	15	x] statt x[] (2 mal)
	16	x statt x] (2 mal)
69	-11	weil diese statt weil sie
94	2	straightinsertion statt straigtinsertion
115	7	a[i-1] statt a[j-1]
	8	a[j+1] statt a[i+1]
	13	k = j+1 ... i-1 statt k = i-1 ... j+1
116	1	relativ statt realtiv
122	-5	einfügen vor Fig. 2.9:
		1. i <= k
	-4	1. streichen
	-1	einfügen vor Fig. 2.11:
		2. k <= i
123	1	2. streichen
	2	einfügen vor Fig. 2.11:
		3. i < k < i
	3	Das statt 3. i < k < i: das

Seite	Zeile	Korrektur
136		in Fig. 2.13 Durchlauf statt Lauf (3 mal)
167		in Fig. 2.17 <- 27 statt -> 27
178	8	p[s,p] statt p[s,p]
188	-12	Die Formel statt Die erste Zeile von
193	11	index = 1..n: statt index 1..n;
225	5,8	pedigree statt Stammbaum
235	8	pf.next := q statt pf.next := p
240	18	sentinel statt tail
247	3	count statt coent
252	-9	dass die Schlüssel statt das die Schlüssel
257	13-15	dass auch sie mittels Rekursion definiert werden können. statt dass sie ... nicht gilt.
262	1	(a+b/c) statt ((a+b)/c)
264	-11	ausgeglichene statt ausgewogene
265	-4	writeln(key); statt writeln;
	-3	h+1 statt h+l
267	14	{stack} statt {stackm
270	13	Routine statt Roetine
	14	die Prozedur orinttree in Program 4.3, die statt Programm 4.3, das
271	4	nil statt nil
274	5	output statt oetput
275	-11,	
	-10	x, pf. statt x.pf.
	-9,-6	count statt coent
278	5	Routine statt Roetine
	21	k1+1 statt k1+l
279	14	0..c4 statt 0..9999
288	11	2*ln(n)/log(n) = 2*ln(2) statt $\lim_{n\to\infty}$ 2*ln(n)/log(n) = 2/ln(2)
292	7, 12	
299	-1	ausgeglichen statt ausgewogen
299		in Fig. 4.35 bezeichne Bilder mit a), b) ... h)
305	1	x statt k
374	-4	How statt Hog
	-4,-1	streiche :

B.G.Teubner Stuttgart

Teubner Studienbücher Fortsetzung

Mathematik Fortsetzung

Jaeger/Wenke: **Lineare Wirtschaftsalgebra**
Eine Einführung
Band 1: XVI, 174 Seiten. DM 18,80 (LAMM)
Band 2: IV, 160 Seiten. DM 18,80 (LAMM)

Kochendörffer: **Determinanten und Matrizen**
IV, 148 Seiten. DM 16,80

Krabs: **Optimierung und Approximation**
208 Seiten. DM 24,80

Stiefel: **Einführung in die numerische Mathematik**
Eine Darstellung unter Betonung des algorithmischen Standpunktes
4. Aufl. 257 Seiten. DM 21,80 (LAMM)

Stummel/Hainer: **Praktische Mathematik**
299 Seiten. DM 26,80

Topsøe: **Informationstheorie**
Eine Einführung. 88 Seiten. DM 11,80

Walter: **Biomathematik für Mediziner**
148 Seiten. DM 14,80

Witting: **Mathematische Statistik**
Eine Einführung in Theorie und Methoden
2. Aufl. 223 Seiten. DM 24,– (LAMM)

Mechanik

Becker: **Technische Strömungslehre**
Eine Einführung in die Grundlagen und technischen Anwendungen der Strömungsmechanik. 3. Aufl. 144 Seiten. DM 12,80

Becker/Piltz: **Übungen zur Technischen Strömungslehre**
120 Seiten. DM 11,80

Becker/Bürger: **Kontinuumsmechanik**
Eine Einführung in die Grundlagen und einfache Anwendungen
228 Seiten. DM 29,– (LAMM)

Magnus: **Schwingungen**
Eine Einführung in die theoretische Behandlung von Schwingungsproblemen. 2. Aufl. 251 Seiten. DM 18,80 (LAMM)

Magnus/Müller: **Grundlagen der Technischen Mechanik**
300 Seiten. DM 25,80 (LAMM)

Müller/Magnus: **Übungen zur Technischen Mechanik**
292 Seiten. DM 25,80 (LAMM)

Wieghardt: **Theoretische Strömungslehre**
Eine Einführung. 2. Aufl. 237 Seiten. DM 24,– (LAMM)

Preisänderungen vorbehalten

Leitfäden der angewandten Mathematik und Mechanik LAMM

Eine Einführung in die Theorie und Praxis fundamentaler Algorithmen und der Darstellung von Daten, mit Ausführungen über die Konkretisierung von Algorithmen und Datenstrukturen zu Computerprogrammen, für Informatiker, Mathematiker und Ingenieure

Fundamentale Datenstrukturen: Das Feld (Array), der Verbund (Record), die Menge (Set)

Die sequentielle Datei (File)

Sortieren am Array

Sortieren von Files

Rekursive Algorithmen

»Backtracking«-Algorithmen

Dynamische Datenstrukturen

Listenverarbeitung mit Zeigervariablen

Bäume, ausgewogene Bäume, B-Bäume

Optimale Suchbäume

Streuspeicherverfahren

B.G.Teubner Stuttgart

ISBN 3-519-02330-X